IBM Developer Kit and Runtime Environment, Java 2
Technology Edition, Version 1.4.2

# Diagnostics Guide

SC34-6358-06

IBM Developer Kit and Runtime Environment, Java 2
Technology Edition, Version 1.4.2

# Diagnostics Guide

SC34-6358-06

**Note**
Before using this information and the product it supports, read the information in Appendix L, "Notices," on page 511.

**Tenth Edition (November 2006)**

This edition applies to all the platforms that are included in the IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 1.4.2 and to all subsequent releases and modifications until otherwise indicated in new editions. Technical changes that have been made since the previous edition of this book are indicated by a vertical bar to the left of each change.

# Contents

# Figures

# Tables

# About this book

This book describes 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.

Both the JRE and the SDK include 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 or AIX®.

This book describes problem determination and diagnostics for the JVM. When you see the terms SDK or JRE, they refer to the JVM only.

## Who should read this book

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

## Before you read this book

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

## How to read this book

This book is to be used with the IBM SDK 1.4.2.

Check the full version of your installed JVM. If you do not know how to do this, see Chapter 12, “First steps in problem determination,” on page 97. Ensure that your JVM is at Version 1.4.2. Some of the diagnostic tools described in this book apply only to this version or later.

You can use this book in three ways:

- As an overview of how the IBM JVM 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 JVM. 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 tools and sample code to which this book refers, see:

  http://www.ibm.com/developerworks/java/jdk/diagnosis/index.html
- For the latest tools and documentation, see IBM developerWorks at:

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

  http://java.sun.com/products/jdk/1.4/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 95 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 81 of the book.

## Conventions and terminology used in this book

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

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

Functions and methods are shown in a monospaced font. For example:

- `ResetJavaVM()`
- `QueryJavaVM()`

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. 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 81.

## Contributors to this book

This book has been put together by members of the IBM Java Technology Center development and service departments in Hursley, Bangalore, Austin, Toronto, and others, including:

Ajjaiah B M
Amar Devegowda
Eduardo Angel
Chris Bailey
John Barfield
Alan Beasley
Flavio Bergamaschi
Bhupesh Gupta
Geoffrey Blandy
Mark Bluemel
Sam Borman
Joe Chacko
Richard Chamberlain
Dave Clarke
Richard Cole
Mike Cotton
Cassius Crockatt
Alan Darlington
Devaprasad K N
Robert Fairley
Ron Fillmore
Ross Grayton
Guruprasad H N
Hari P Venkateshaiah
Lakshmi Shankar

Linda Howard
Steve Hughes
Clive Kates
Matthew Kilner
Sripathi Kodi
Roger Leuckie
Nigel Lewis
Bob Maddison
Neil Masson
Mahesh P Kumar
Wai-Kau Mak
Caroline Maynard
Diego Oriato
Panneer S Gangatharan
Mark Partridge
Pavan Kumar B
Prasanna K Kalle
Prashanth K N
Rajeev Palanki
Rajesh Kumar J
David Reynolds
Neil Richards
Phil Rosenthal
Ruchika Gupta
Rupesh B Khandekar
David Screen
Sreekanth R Iyer
Jon Stone
Sudarshan Rao
Subramanian V Ganesh
Thekkepat A Vinod
Venkat R Vellaisamy
Venugopal Kailaikurthi
Phil Vickers
Chris White

## Summary of changes for Version 1.4.2, Service Refresh 7

This book has been updated to include minor changes that apply to the IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 1.4.2. No significant changes were made to this book for Service Refresh 7. This edition of the book (SC34-6358–06) was produced in November 2006.

## Summary of changes for Version 1.4.2, Service Refresh 6

This book has been updated to include minor changes that apply to the IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 1.4.2. No significant changes were made to this book for Service Refresh 6. This edition of the book (SC34-6358–05) was produced in August 2006.

## Summary of changes for Version 1.4.2, Service Refresh 5

This book was updated to include changes that apply to the IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 1.4.2. Technical changes that have been made since the previous edition of this book (SC34-6358–03) produced in January 2006, are indicated by a vertical bar to the left of each change.

The major changes and additions are:

- A major revision of Appendix I, "Using the alternative JVM for Java debugging," on page 499 now that the alternative debug environment applies to AIX and Linux PPC32 and PPC64.

## Summary of changes for Version 1.4.2, Service Refresh 4

The major changes and additions are:

- A new chapter, Chapter 37, "Using DTFJ," on page 375
- ALLOCATION_THRESHOLD environment variable, see Table 33 on page 410.
- JVMDBG messages, see "JVM error messages for JVMDBG" on page 439.
- Increased process space under windows, see "Windows 32-bit large address aware support" on page 152.

## Summary of changes for Version 1.4.2, Service Refresh 3

The major changes and additions are:

- A revised section ″Threading libraries″ and a new section ″Floating stacks″ in Chapter 15, "Linux problem determination," on page 129.
- Further changes in Chapter 26, "Using Heapdump," on page 245.
- New JVMHP messages in "JVM error messages for JVMHP" on page 456.
- Small changes to the **-Xpd** option in Appendix J, "Using a Problem Determination build of the JVM," on page 503.

## Summary of changes for Version 1.4.2, Service Refresh 2

This Diagnostics Guide now includes items in the Addenda file up to June 2005 and there are some changes to CORBA properties and minor codes.

## Summary of changes for Version 1.4.2

The major changes and additions are:

- Chapter 17, "Hewlett-Packard SDK problem determination," on page 149 is new.
- "How to write a JVMMI Heapdump agent" on page 249 is new.
- jformat information in "Opening the dump" on page 264 and following sections is new.
- Backtrace information in "Controlling the trace" on page 323 is new.

- Some triggered trace information is new in "Controlling the trace" on page 323.
- "Sample JVMMI Heapdump agent" on page 352 is new.
- "Universal Trace Engine error messages" on page 474 is new.
- The **-Xifa** parameter in Appendix G, "Command-line parameters," on page 487 is new.
- Appendix J, "Using a Problem Determination build of the JVM," on page 503 is new. The new build replaces the _g builds.

## Summary of changes for the Version 1.4.1 April 2004 update

The main changes and additions were:

- Chapter 2, "Understanding the Garbage Collector," on page 7 contains new sections: "Subpool (AIX only)" on page 18, "Avoiding fragmentation" on page 22, and "Frequently asked questions about the Garbage Collector" on page 27.
- Chapter 3, "Understanding the class loader," on page 31 contains an expanded introduction and new sections: "Eager and lazy loading" on page 31 and "WebSphere 5.0 ClassLoader overview" on page 35.
- Chapter 7, "Understanding Java Remote Method Invocation," on page 77 is new.
- Chapter 14, "AIX problem determination," on page 101 contains a number of enhancements.
- Chapter 18, "Windows problem determination," on page 151 contains new sections: "Creating a user dump file for a hung process using the Dr. Watson utility" on page 160 and "Controlling the JVM when used as a browser plug-in" on page 165.
- Chapter 19, "z/OS problem determination," on page 167 contains new sections: one about HPI trace (not valid for 1.4.2) and "Working with TDUMPs using IPCS" on page 176.
- Chapter 30, "JIT diagnostics," on page 295 contains a new section: "Performance of short-running applications" on page 298.
- Chapter 31, "Garbage Collector diagnostics," on page 299 contains updated -X options.
- Appendix E, "Environment variables," on page 407 contains new entries.
- Appendix G, "Command-line parameters," on page 487 contains new entries.
- Appendix K, "Some notes on jformat and the jvmdcf file," on page 505 is new.

## Summary of changes for Version 1.4.1, Service Refresh 1

This update showed the changes that applied to the IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 1.4.1, Service Refresh 1.

The major changes were:

- The addition of a chapter that describes how to diagnose the class loader
- The addition of an appendix that describes how to diagnose the WebSphere workbench runtime environment
- The addition of an appendix that describes the most-common CORBA minor codes
- Major revision to Heapdump information

## Summary of changes for Version 1.4.1

This update showed the changes that applied to the IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 1.4.1.

The major changes were:

- The addition of chapters for AS/400® and OS/2® problem determination
- The addition of a chapter that describes the Java Native Interface (JNI)
- Updates, deletions, and additions to the JVM messages
- Major revision to ORB information
- Major revision to tracing information
- Major revision to Heapdump information

# Part 1. Understanding the IBM JVM

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 JVM

The IBM Java Virtual Machine (JVM) is the core component of the IBM Java Runtime Environment (JRE). The IBM JRE includes the JVM, the class libraries (including the IBM ORB), and other files that provide the runtime support that is necessary for a Java application stack.

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

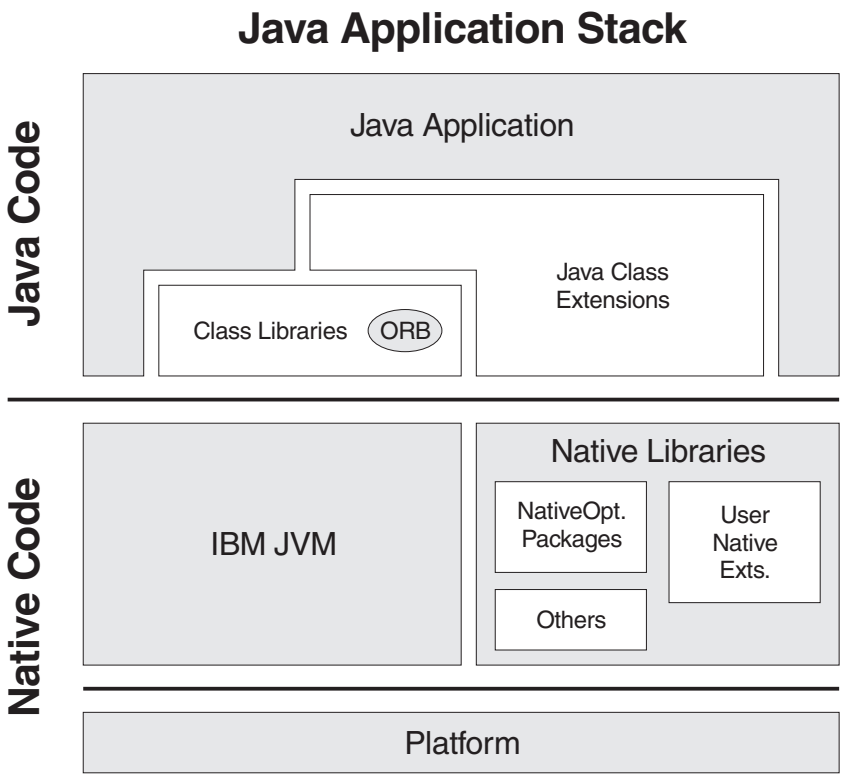



*Figure 1. The components of a typical Java Application Stack and the IBM JRE*

The IBM Java Virtual Machine (JVM) technology consists of a set of subcomponents (building blocks). Each subcomponent defines a high-level logical grouping of functions in the IBM JVM. The core IBM JVM is built with the following set of default subcomponents that provides a compatible Java Virtual Machine.

- Core interface
- Execution management
- Execution engine
- Diagnostics
- Class Loader
- Data conversion
- Locking
- Storage
- Hardware platform interface

Figure 2 shows subcomponent structure of the IBM JVM.

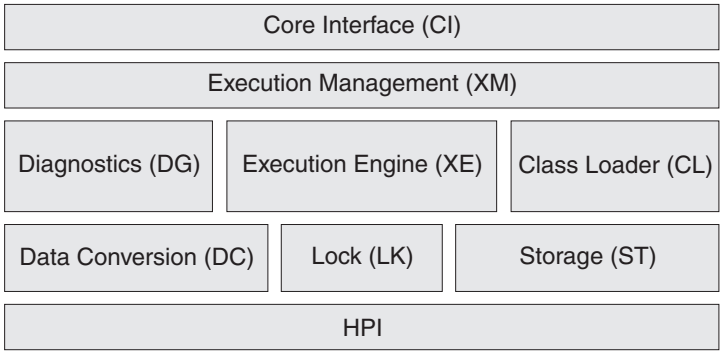



*Figure 2. Subcomponent structure of the IBM JVM*

Subcomponents are built around a collection of functional units. Each functional unit defines a lower-level logical grouping of functions.

## Core interface

This subcomponent encapsulates ***all*** interaction with the user, external programs, and operating environment. It is responsible for initiation of the JVM. It also:

- Provides presentation (but ***not*** execution) of all external APIs (for example, JNI, JVMDI, JVMPI)
- Presents the HPI APIs to other Java2 components
- Processes command-line input
- Converts relevant environmental settings to platform-neutral initiation information
- Provides internal APIs to enable other subcomponents to interact with the console
- Holds routines for interacting with the console; nominally, standard in, out, and err
- Provides support for issuing formatted messages that are suitable for NLS
- Holds routines for accessing the system properties

## Execution engine (XE)

This subcomponent provides all methods of executing Java byte codes, both compiled and interpretive. It:

- Executes the byte code (in whatever form)
- Calls native method routines
- Contains and defines byte code compiler (JIT) interfaces
- Provides support for math functions that the byte code requires
- Provides support for raising Java exceptions

## Execution management (XM)

This subcomponent provides process control and management of multiple execution engines. Is initiated by the core interface. It provides:

- Threading facilities
- Runtime configuration; setting and inquiry
- Support for raising internal exceptions

- End JVM processing
- Support for the resolution and loading of native methods

## Diagnostics (DG)

This subcomponent provides all diagnostic and debug services and facilities. It is also responsible for providing methods for raising events. It provides:
- Support for issuing events
- Implementation of debug APIs
- Trace facilities
- Reliability, availability, and serviceability (RAS) facilities
- First failure data capture (FFDC) facilities

## Class Loader (CL)

This subcomponent provides all support functions to Java classes, except the execution. This includes:
- Loading
- Resolution
- Verification
- Initialization
- Methods for interrogation of class abilities
- Implementation of reflection APIs

## Data conversion (DC)

This subcomponent provides support for converting data between various formats. This includes:
- UTF Translation
- String conversion
- Support for primitive types

## Lock (LK)

This subcomponent provides locking and synchronization services.

## Storage (ST)

This subcomponent encapsulates all support for storage services. It provides:
- Facilities to create, manage, and destroy discrete units of storage
- Specific allocation strategies
- The Java object store (garbage collectable heap)

## Hardware platform interface (HPI)

This subcomponent consists of a set of well-defined functions that provide low-level facilities and services in a platform-neutral way. The HPI is an external interface that is defined by Sun.

# Chapter 2. Understanding the Garbage Collector

This chapter describes the Garbage Collector under these headings:



For detailed information about diagnosing Garbage Collector problems, see Chapter 31, "Garbage Collector diagnostics," on page 299.

For reference information about the Garbage Collector command-line parameters, see "Garbage Collector command-line parameters" on page 491.

For more information about the workings of the Garbage Collector, see *IBM JVM Garbage Collection and Storage Allocation Techniques* at http://www.ibm.com/developerworks/java/jdk/diagnosis/index.html.

## Overview of garbage collection

Many users have difficulty understanding the Garbage Collector. This chapter provides:

- A summary of some of the diagnostic techniques that are described elsewhere in this book
- Knowledge of how 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 objects, arrays, and classes. 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, arrays, or classes. 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 the end of this cache without the need to grab the heap lock. Therefore, the allocation is very efficient.

Objects that are allocated through this interface are, if small enough (currently 512 bytes), allocated from the cache. This cache is often referred to as the *thread local heap* or TLH.

## 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 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 get all the locks that the garbage collection process needs. This step ensures that other threads are not suspended while they are holding critical locks. All the other threads are then suspended. 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 vector that defines live objects.

### Sweep phase

After the mark phase, the mark vector contains a bit for every reachable object that is in the heap. The mark vector must be a subset of the allocbits vector. The task of the sweep phase is to identify the intersection of these vectors; that is, objects that have been allocated but are no longer referenced.

The original technique for this sweep phase was to start a scan from the bottom of the heap, and visit each object in turn. The length of each object was held in the word that immediately preceded it on the heap. At each object, the appropriate allocbit and markbit was tested to locate the garbage.

Now, the *bitsweep* technique avoids the need to scan the objects that are in the heap and therefore avoids the associated overhead cost for paging. In the bitsweep technique, the mark vector is examined directly to look for long sequences of zeros (not marked), which probably identify free space.

When such a long sequence is found, the length of the object that is at the start of the sequence is examined to determine the amount of free space that is to be released. Objects are not normally allocated from the heap itself but from thread local heap, which is allocated from the heap and later used by an individual thread to meet any allocation requirements.

### 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. Because compaction can take a long time, the Garbage Collector tries to avoid it if possible. Compaction is, therefore, a rare event. For more information, see "Compaction avoidance" on page 17.

## Heap size

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

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

**OS/390® and AIX®**
64 MB.

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

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

**OS/390**
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 **verbosegc** to help you monitor the heap.

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

**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 **verbosegc** trace shows whether finalizers are being called.
- Avoid compaction. A **verbosegc** 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.

### The system heap

The system heap contains only objects that have a life expectancy of the life of the JVM. The objects that are in this heap are the class objects for system and shareable middleware, and for application classes. The system heap is never garbage collected because all objects that are in it either are reachable for the lifetime of the JVM, or, in the case of shareable application classes, have been selected to be reused during the lifetime of the JVM. The system heap is a chain of noncontiguous areas of storage. The initial size of the system heap is 128 KB in 32-bit architecture, and 8 MB in 64-bit architecture. If this fills, the system heap obtains another extent and chains the extents together.

## 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 occurs when the allocation request is greater than 512 bytes or when the allocation cannot be contained 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 the end of this cache without the need to grab the heap lock; therefore, cache allocation is very efficient.

The Garbage Collector uses cache allocation if the size of the object is less than 512 bytes, or if the object can be contained in the existing cache.

The cache block is sometimes called a thread local heap (TLH). The size of the TLH varies from 2 KB to 164 KB, depending on the use of the TLH.

## The wilderness

The wilderness is now called the Large Object Area (LOA). The way in which it works has changed to improve the allocation of large objects. The terms "wilderness", "Large Object Area", and "LOA" are used interchangeably throughout the remainder of this book.

### Initialization

The LOA boundary is calculated when the heap is initialized, and recalculated after every garbage collection. The initial size of the LOA is 5% of the current heap size. It can then be readjusted as follows:

- If the free space size and the LOA size combined provide less space than is available when the -Xminf value (default 30%) of the heap is free, the LOA size is zero.
- If the free space size provides less space than is available when the -Xminf value (default 30%) of the heap is free, the LOA size will be reduced so that the free space size equals the -Xminf value.

When the Garbage Collector calculates the size of the LOA, it also sets ca_progressFreeObjectCtr to be equal to the free space size minus the -Xminf value of the current heap size. This variable is then used to decide when to allocate out of the LOA.

### Expansion and shrinkage

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

- If an allocation failure occurs on the main heap:
  - If the current size of the heap is greater than the 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 heap is equal to or less than the initial size, and if the amount of free space in the LOA is greater than 90%:
    - If the current size of the heap is greater than 1%, reduce by 1% the percentage of space that is allocated to the LOA.
    - If the current size of the heap is 1% or less, reduce by 0.1%, to a minimum of 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 5 times the current size of the LOA, increase the LOA by 1% to a maximum of 20%.
  - If the current size of the heap is less then the 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 heap is equal to or greater than the initial size, and if the amount of free space in the LOA is less than 30%, increase the LOA by 1% to a maximum of 20%.

This algorithm enables the Garbage Collector to expand the LOA if the LOA is being highly used, and shrink it if it is being lightly used, or not used at all. If the usage changes, the Garbage Collector tries to get the LOA back to 5%. If two expansions occur without an intervening shrinkage, the Garbage Collector triggers an incremental compaction by using the trigger COMPACT_LOA_EXPANDED.

### Allocation in the LOA

Allocation occurs before and after a garbage collection.

**Before a garbage collection:** Before a garbage collection, allocation from the LOA is done in manageAllocFailure(), which is called after the Garbage Collector has failed to allocate from the free list in either heap lock allocation or cache allocation. At this time, storage is released only from the first half of the LOA. The Garbage Collector releases storage for either of two reasons:

- If the size of the request is equal to, or greater than, 64 KB.
- If the free space is greater than ca_progressFreeObjectCtr, the Garbage Collector has not made enough allocation progress, so it tries to find space in the LOA.

In both cases, if the Garbage Collector finds space in the LOA, it puts the free chunk at the beginning of the free list, and returns without initiating a garbage collection.

**After a garbage collection:** The second half of the LOA is used to allocate objects after a garbage collection. In handleFreeChunk, if the only chunk that is large enough to satisfy the allocation request is in the LOA, the Garbage Collector splits the chunk and releases enough storage for the request. If three consecutive releases of storage come from the LOA in this way, the Garbage Collector triggers an incremental compaction by using the trigger COMPACT_LOA_PRESSURE.

## Pinned clusters

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.

The Garbage Collector allocates a *kCluster* as the first object at the bottom of the heap. A kCluster is an area of storage that is used exclusively for class blocks. It is large enough to hold 1280 entries. Each class block is 256 bytes long.

The Garbage Collector then allocates a *pCluster* as the second object on the heap. A pCluster is an area of storage that is used to allocate any pinned objects. It is 16 KB long.

When the kCluster is full, the Garbage Collector allocates class blocks in the pCluster. When the pCluster is full, the Garbage Collector allocates a new pCluster of 2 KB. Because this new pCluster can be allocated anywhere, it can cause problems.

To remove these problems, the pinnedFreeList changes the way in which the pCluster is allocated. The concept is that after every garbage collection, the Garbage Collector takes an amount of storage from the bottom of the free list and chains it from the pinnedFreeList. Allocation requests for pClusters use the pinnedFreeList, while other allocation requests use the free list. When either free list is exhausted, the Garbage Collector causes an allocation failure and a garbage collection. This action ensures that all pClusters are allocated in the lowest-available storage location in the heap.

The Garbage Collector uses this algorithm to determine how much storage to put on the pinnedFreeList:

- The initial allocation is for 50 KB.
- If this is not the initial allocation and the pinnedFreeList is empty, the Garbage Collector allocates 50 KB or five times the amount of allocations from the clusters since the last garbage collection, whichever is the larger.

- If this is not the initial allocation and the pinnedFreeList is ***not*** empty, the Garbage Collector allocates 2 KB or five times amount of allocations from the clusters since the last garbage collection, whichever is the larger.

This algorithm increases the amount of storage that is available when the application is loading many classes. It therefore avoids an allocation failure that is due to an exhausted pinnedFreeList. It also reduces the amount of storage that is on the pinnedFreeList when little allocation of pinned clusters exists, and therefore avoids the need to remove large amounts of storage from the free list.

The buildPinnedFreeList function builds the pinnedFreeList by using the above algorithm. This function is called from the following places:

- In initializeClusters
- At the end of expandHeap
- At the end of gc0_locked

The Garbage Collector makes allocations from the pinnedFreeList by calling the nextPinnedCluster function. This function works in a way that is similar to the way in which nextTLH works; that is, it always takes the next available free chunk on the pinnedFreeList. If the pinnedFreeList is empty, it calls manageAllocFailure.

In realObjCAlloc, if no room remains in the clusters, the Garbage Collector calls nextPinnedCluster to allocate a new pCluster.

In initializeClusters, the Garbage Collector calls nextPinnedCluster, which allocates an initial pCluster of 50 KB because 50 KB is the size of the only free chunk that is on the pinnedFreeList. The free chunk has that size because the pinnedFreeList had the initial allocation of 50 KB.

## 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. It first gets all the locks that are required for a garbage collection, then suspends all the other threads. Garbage collection then goes through the three phases: mark, sweep, and, optionally, compaction. The IBM Garbage Collector is a stop-the-world (STW) operation, because all application threads are stopped while the garbage is collected.

### Conservative and type-accurate garbage collection

A Garbage Collector is allowed, by the JVM specification, to be either conservative or type accurate. The terms relate to the way pointers to objects are handled. A type-accurate Garbage Collector can determine whether a pointer to an object really is a pointer or whether it is only application data that happens to look like a pointer to an object. Conservative collectors cannot determine this.

All Garbage Collectors have to find a root set of object pointers from which they can trace all other objects. The IBM Garbage Collector handles pointers to the root set conservatively. This means that, although objects in the root set are subject to collection, they cannot be moved. If they were moved (in a heap compaction), the Garbage Collector would have to reset the pointer to the root object, which might be application data. Except for the root set, all other objects are traced type-accurately.

## 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 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 instances of 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. The Garbage Collector views the stack of a thread as a set of 4-byte fields (8 bytes in 64-bit architecture) and scans them from the top to the bottom of each of the stacks. The Garbage Collector assumes that the stacks are 4-byte aligned (8-byte aligned in 64-bit architecture). Each slot is examined to see whether it points at an object that is in the heap. Note that this does not make it necessarily a pointer to an object, because it might be only an accidental combination of bits in a float or integer. So, when the Garbage Collector performs the scan of a thread stack, it handles conservatively anything that it finds. Anything that points at an object is assumed to be an object, but the object in question must not be moved during garbage collection. A slot is thought to be a pointer to an object if it meets these three requirements:

1. It is grained on an 8-byte boundary.
2. It is inside the bounds of the heap.
3. The allocbit is on.

Objects that are referenced in this way are known as roots and have their dosed bit set on, to indicate that they cannot be moved in any later compaction phase. Tracing can now proceed accurately. That is, the Garbage Collector can find references in the roots to other objects and, because it knows that they are real references, it can move them during compaction because it can change the reference. The tracing process uses a stack that can hold 4 KB entries. All references that are pushed to the stack are marked at the same time by setting the relevant markbit on. The roots are marked and pushed to the stack and then the Garbage Collector starts to pop entries off the stack and trace them. Normal objects (not arrays) are traced by using the classblock, which tells where references to other objects are to be found in this object. As each reference is found, if it is not already marked, it is marked and pushed.

Array objects are traced by looking at each array entry and, if it is not already marked, it is marked and pushed. Some additional code traces a small portion of the array at a time, to try to avoid mark stack overflow.

The above process continues repeatedly until the mark stack eventually becomes empty.

### Mark stack overflow

Because the mark stack has a fixed size, it can overflow. If this occurs, the Garbage Collector:

- Sets a global flag to indicate that mark stack overflow has occurred
- Sets the `NotYetScanned` bit in the object that could not be pushed

Tracing can then continue with all other objects that could not be pushed because they have their `NotYetScanned` bit set. When all tracing is complete, the Garbage

Collector then walks the heap by starting at the first object and using the size field to navigate to the next object. All found objects that have their `NotYetScanned` bit set are marked and pushed to the mark stack. The `NotYetScanned` bit is set off and tracing continues as before. It is possible to get another mark stack overflow, in which case the Garbage Collector must go through the whole process again until all reachable objects are marked.

## Parallel Mark

With Bitwise Sweep and Compaction Avoidance, the majority of garbage collection time is spent marking objects. Therefore, a parallel version of Garbage Collector Mark has been developed. 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 and a facility that shares work between those threads. Parallel Mark still requires the participation of one application thread, which is used as the master coordinating agent. This thread performs very much as it always did, including the responsibility for scanning C-stacks to identify root pointers for the collection. A platform with N processors also has N-1 new helper threads, which work with the master thread to complete the marking phase of garbage collection. The default number of threads can be overridden with the **-Xgcthreads** parameter. A value of 1 results in no helper threads. The **-Xgcthreads** option accepts any value greater than 0, but you gain little by setting it to more than N-1.

At a high level, each marker thread is provided with a local stack and a sharable queue, both of which contain references to objects that are marked but not yet scanned. Threads do most of the marking work by using their local stacks, synchronizing on sharable queues only when work balance requires it. Mark bits are updated by using atomic primitives that require no additional lock. Because each thread has a Mark Stack that can hold 4 KB entries, and a Mark Queue that can hold 2 KB entries, the chances of a Mark Stack Overflow are reduced.

## 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 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. It is the same write barrier that is required by resettable, as described later. 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 for two reasons: a write to a byte is quicker than a bit change, and the other bits are reserved for future use. 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 scans all roots, uses the marked cards to see what must be retraced, 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.

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.

This parameter enables concurrent mark:

**-Xgcpolicy:**<*optthruput* | *optavgpause*>

Setting **-Xgcpolicy** to *optthruput* disables concurrent mark. If you do not have pause time problems (as seen by erratic application response times), you get the best throughput with this option. *Optthruput* is the default setting. 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.

## Sweep phase

After the mark phase, the markbits vector contains a bit for every reachable object that is in the heap, and must be a subset of the allocbits vector. The sweep phase identifies the intersection of the allocbits and markbits vectors; that is, objects that have been allocated but are no longer referenced. In the bitsweep technique, the Garbage Collector examines the markbits vector directly and looks for long sequences of zeros, which probably identify free space. When such a long sequence is found the Garbage Collector checks the length of the object at the start of the sequence to determine the amount of free space that is to be released. If this amount of free space is greater than 512 bytes plus the header size, this free chunk is put on the freelist. 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. During this process, the markbits are copied to the allocbits so that on completion, the allocbits correctly represent the allocated objects that are on the heap.

### Parallel bitwise sweep

Parallel Bitwise Sweep improves 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 **-Xgcthreadsn** parameter. The heap is divided into sections. The number of sections is significantly larger than the number of helper threads. The calculation for the number of sections is as follows:

- 32 x the number of helper threads, or
- The maximum heap size ÷ 16 MB

whichever is larger. The helper threads take a section at a time and scan it, performing a modified bitwise sweep. The results of this scan are stored for each section. When all sections have been scanned, the freelist is built.

## 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. If one of those references was from a stack, and therefore the Garbage Collector is not sure that it was an object reference (it might have been a float, for example), the Garbage Collector cannot move the object. Such objects that are temporarily fixed in position are referred to as *dosed* in the code and have the dosed bit set in the header word to indicate this fact. Similarly, objects can be *pinned* during some JNI operations. Pinning has the same effect but is permanent until the object is explicitly unpinned by JNI. Objects that remain mobile are compacted in two phases by taking advantage of the fact that the mptr is known to have the low three bits zero and unused. One of these bits can therefore be used to denote the fact that it has been swapped. Note that this swapped bit is applied in two places: the size + flags field (where it is known as OLINK_IsSwapped) and also the mptr (where it is known as GC_FirstSwapped). In both cases, the least significant bit (x01) is being set.

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. Pinned and dosed objects that cannot be moved make the picture difficult, but do not change the general idea.

## Compaction avoidance

Compaction avoidance focuses on correct object placement. It therefore reduces, and in many cases removes, the need for compaction. An important point of this approach is a concept that is called Wilderness Preservation. Wilderness Preservation attempts to keep a region of the heap in an unused state by focusing allocation activity elsewhere. It does this by making a boundary between most of the heap and a reserved wilderness portion. In typical cases, noncompacting garbage collection events are triggered whenever the wilderness is threatened. The wilderness is consumed (eroded) only when necessary to satisfy a large allocation, or when not enough allocation progress has been made since the previous garbage collection.

The wilderness is allocated at the end of the active part of the heap. Its size is 5% of the active part of the heap, with a maximum of 3 MB. On heap lock allocation failure, if enough allocation progress has been made since the last garbage collection, the Garbage Collector runs. Enough progress means that at least 30% of the heap has been allocated since the last garbage collection. This is the default. It

can be changed with the **-Xminf** parameter. If not enough progress has been made, the allocation is immediately satisfied from the wilderness if possible. Otherwise, a normal allocation failure occurs. Not enough progress has been made if the Garbage Collector gets an allocation request for a large object that cannot be satisfied before the free list is exhausted. In this condition, the reserved wilderness can satisfy the request, and avoid a garbage collection and a compaction.

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.
- At least half the previously available memory has been consumed by TLH allocations (ensuring an accurate sample) and the average TLH size falls below 1000 bytes.
- Less than 5% of the active heap is free.
- Less than 128 KB of the active heap is free.

## Subpool (AIX only)

On AIX only, release 1.4.1 Service Refresh 1 introduced an improved GC policy for object allocation that is specifically targeted at improving the performance of object allocation. You invoke it with the **-Xgcpolicy:subpool** command-line option.

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

## Reference objects

Reference objects enable all references to be handled and processed in the same way. Therefore, the Garbage Collector creates two separate objects on the heap: the object itself and a separate reference object. The reference objects can optionally be associated with a queue to which they will be added when the referent becomes unreachable. Instances of SoftReference, WeakReference, and PhantomReference are created by the user and cannot be changed; they cannot be made to refer to other than the object that they referenced on creation. Objects that are associated with a finalizer are 'registered' with the Finalizer class on creation. The result is the creation of a FinalReference object that is associated with the Finalizer queue and refers to the object that is to be finalized.

During garbage collection, these reference objects are handled specially; that is, the referent field is not traced during the marking phase. When marking is complete, the references are processed in sequence:

1. Soft

2. Weak
3. Final
4. Phantom

Processing of SoftReference objects is specialized; that is, the Garbage Collector can decide that these references should be cleared if the referent is unmarked (unreachable except for a path through a reference). The clearing is done if memory is running out and is done selectively on the principle of most recent usage. Usage is measured by the last time that the get method was called, which can give some unexpected, although valid, results. When a reference object is being processed, its referent is marked, ensuring that when, for example, a FinalReference is processed for an object that also has a SoftReference, the FinalReference sees a marked referent. The FinalReference, therefore, is not queued for processing. The result is that references are queued in successive garbage collection cycles.

References to unmarked objects are initially queued to the ReferenceHandler thread that is in the reference class. The ReferenceHandler takes objects off its queue and looks at their individual **queue** field. If an object is associated with a specific queue, it is requeued to it for further processing. Therefore, the FinalReference objects are requeued and eventually their finalize method is run by the finalizer 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 and when all the threads have been restarted, but the HEAP_LOCK 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 13% of the time is being spent in garbage collection, and is expanding by the minimum expansion amount. (**-Xmine**) does not result in a heap that is greater than the maximum percentage of free space (**-Xmaxf**).

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 because the Garbage Collector did not free enough storage and the JVM is not spending more than 13% in garbage collection, the heap is expanded by the allocation request.
- If the heap is expanding for any other reason, the Garbage Collector calculates how much expansion is needed to get 17.5% free space. This is adjusted as above, depending on **-Xmaxe** and **-Xmine**.
- Finally, the Garbage Collector must ensure 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 64 KB boundary on 32-bit architecture, or a 4 MB boundary on 64-bit architecture.

## Heap shrinkage

Heap shrinkage occurs after garbage collection, but when all the threads are still suspended. 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 64 KB boundary on 32-bit architecture, or a 4 MB boundary on 64-bit architecture.

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

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

### Resettable JVM (z/OS only)

The resettable JVM is available only on z/OS™.

You can find documentation about the Resettable JVM in *New IBM Technology featuring Persistent Reusable Java Virtual Machines*, SC34-6034-01. This is available at http://www.s390.ibm.com/Java

## 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 verbosegc 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 most likely to be very fragmented when a need to do a heap compaction occurs. 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 exits.
   If it cannot find enough garbage, it goes to the next step.
2. The Garbage Collector runs compaction.
   If it cannot find enough garbage, it goes to the next step.
3. 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 and when the heap is full of live objects and cannot be compacted, the Garbage Collector expands the heap size.

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.

## Avoiding fragmentation

For a large Java application, such as WebSphere Application Server, the default kCluster space (for an introduction to kCluster, see "Pinned clusters" on page 12) might not be sufficient to allocate all classblocks. Use the **-Xk** and **-Xp** command-line options to specify kCluster and pCluster sizes. For example:

```
-Xknnnn
```

where `nnnn` specifies the maximum number of classes the kCluster will contain. **-Xk** instructs the JVM to allocate space for nnnn class blocks in kCluster.

GC trace data obtained by setting **-Dibm.dg.trc.print=st_verify** (for more information about GC tracing, see "Tracing" on page 308) provides a guide for the optimum value of the nnnn parameter. For example::

```
<GC(VFY-SUM): pinned=4265(classes=3955/freeclasses=0)
dosed=10388 movable=1233792 free=5658>
```

The 'pinned/classes' size is about the correct size needed for the **-Xk** parameter. You are recommended to add 10% to the reported value (3955). So, in this example, `-Xk4200` would be a good setting.

The difference between `pinned` (=4265) and `classes` (=3955) provides a guide for the initial size of pCluster, although because each object might be different in size it is hard to predict the requirements for the pCluster and pCluster overflow options. You can specify the pCluster and pCluster overflow sizes by the -Xp command-line option:

```
-Xpiiii[K][,oooo[K]]
```

where `iiii` specifies the size of the initial pCluster in KB and `oooo` optionally specifies the size of overflow (subsequent) pClusters in KB. Default values of `iiii` and `oooo` are 16 KB and 2 KB respectively .

Where your application suffers from heap fragmentation, use GC trace and specify the **-Xk** option. If the problem persists, experiment with higher initial pCluster settings and overflow pCluster sizes.

## Using verbosegc

The verbosegc output is fully described in Chapter 31, "Garbage Collector diagnostics," on page 299. Switch on verbosegc 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 parameter) that is needed. If this value is much larger than the defaults (see Appendix H, "Default settings for the JVM," on page 495), think about reducing this value a little to get efficient and rapid compaction up to this value, as described in "Initial and maximum heap sizes" on page 21.

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

### Using fine tuning options

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

-Xmaxe
-Xmine
-Xmaxf
-Xminf

These are described in "Heap expansion" on page 19 and "Heap shrinkage" on page 20.

## 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 unused objects.
   a. The Garbage Collector does not guarantee to find all unused objects.
2. The Garbage Collector will not collect live 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 a memory fault occurs.
   b. The Garbage Collector will honor manual invocations.
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

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

Note clause 4b. 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".

## How to coexist with the Garbage Collector

### Predicting Garbage Collector behavior

Why would you want to predict the behavior of the Garbage Collector? Java service often receive PMRs that are implicitly expecting predictable behavior. The IBM Garbage Collector does not have predictable behavior. The following sections

describe why. This information is important in helping you to understand some of the remaining advice that is given in this section.

Consider the initial conditions that face the Garbage Collector when it starts a cycle. The IBM Garbage Collector is not completely type-accurate. This means that no formal way exists to distinguish objects, or references to objects, from normal data. Some JVMs have complete type accuracy. The IBM JVM does not. So how does it find objects from which it can start tracing the graph of live objects ?

The Garbage Collector scans all the stacks and registers of running threads and also scans a known area where JNI references are stored. If a number is found that looks like it might be a reference to the Java heap (that is, it points to an object), the Garbage Collector follows the link and handles the resultant data as an object. The set of objects that is found in this way is known as the root set. When a root object has been found, the Garbage Collector checks what should be object references from that object. If these references are all valid, most likely the root object is an object, and it is handled as such. If these references are not all valid, the object is discarded from the root set. Any references from a root object are handled type accurately.

A small but finite chance exists that some application datum on the stack, or in a register, is not actually an object reference, but coincidentally looks like one. This has two important implications:

1. The Garbage Collector follows an invalid reference into the heap and traces from that reference a graph of objects that are considered reachable and, therefore, not garbage. If some or all of those objects really are garbage, they are not collected. This is known as retained garbage (see clause 1a on page 23). This is unavoidable with the IBM Garbage Collector. In normal conditions, it is not expected that consecutive garbage collection cycles would throw up the same invalid reference, so retained garbage will be collected eventually (see clause5 on page 23).
2. The root set of objects are treated conservatively. This means that they are not moveable. If the garbage collection cycle invokes a heap compaction, the Garbage Collector cannot move these objects, because it would then change the reference on the stack or register, and this might be an application datum. Therefore, the set of root objects, which can be quite large, are unmovable in the same garbage collection cycle. Obviously, the root set are considered reachable and also noncollectable.

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 23). This has significant consequences for finalizers (clause 7 on page 23), which are described more fully in “Finalizers” on page 25.

## 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 programme 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:

- The objects are in TLH and do not become visible until the TLH flushes.
- 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 23. True garbage is always found eventually, but it is not possible to predict when (clause 5 on page 23).

## 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 23. These topics are examine 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 class 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 makes a list of all 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 task adds more overhead into the garbage collection cycle.

### Finalizers and the garbage collection contract

Garbage Collector contract clause 7 on page 23, 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 23). 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 23).
- Because a finalized object might be retained garbage, it is possible that a finalizer might not run at all (contract subclause 7c on page 23).

### 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 23. 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 23 notes that the Garbage Collector always honors 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 23), which is within the JVM specification.

## Frequently asked questions about the Garbage Collector

**What are the default heap sizes?**
See "Heap size" on page 9.

**If I don't specify -Xmx and -Xms, what values will Java use?**
See Appendix H, "Default settings for the JVM," on page 495.

**What are default values for the native stack (-Xss) and Java stack (-Xoss)?**
The Native stack size is machine-dependent, because it is based on the platform's C stack usage. The Java stack size is 400*1024

**What is the difference between the GC policies optavgpause and optthruput?**
**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.

**What is the default GC mode (optavgpause or optthruput)?**
**optthruput** - that is, concurrent marking is 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

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.

**Is incremental compaction enabled by default?**
Yes. But incremental compaction works only if the size of the heap is at least 128 MB.

**What is double allocation failure?**
Double allocation failure refers to the condition in which the GC believes that it has freed enough heap storage to satisfy the current allocation request, but still the allocation request fails. This is clearly an error condition, and it results in the JVM closing down with a ″panic″ error message.

**What are pinned and dosed objects?**
Pinned and dosed objects are the immovable objects on the Java heap. GC does not move these objects during compaction. These are the major cause of heap fragmentation.

All objects that are referenced from JNI are pinned. All objects on the heap that are referenced from the thread stacks are dosed.

**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**.
- If the application uses JNI, make sure JNI references are properly cleared. All objects being referenced by JNI are pinned and not moved during compaction, contributing significantly to heap fragmentation.

**Does running with -Xpartialcompactgc avoid heap fragmentation?**
This option can be useful when used with incremental compaction and will reduce fragmentation. However, it is no more effective than regular compaction for pinned and dosed objects.

**What is Mark Stack Overflow? Why is MSO bad for performance?**
Mark stacks 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. Mark stacks are of fixed size, so they 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?**
There is nothing an application can do to avoid MSO, except to reduce the number of objects it allocates. The following suggestions are not guaranteed to avoid MSO:

- Increase the number of GC helper threads using **-Xgcthreads** command-line option
- Decrease the size of the Java heap using the **-Xmx** setting.
- Use a small initial value for the heap or use the default.

**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).

**How can I confirm if the OutOfMemoryError was caused by the Java heap becoming exhausted?**

Run with the **-verbosegc** option. VerboseGC will show messages such as `Insufficient heap space to satisfy allocation request` when the Java heap is exhausted

**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.

**What does verifyHeap do? What can we learn about the problem or crash from verifyHeap?**

The **verifyHeap** option can verify the integrity of the heap and free list during the phase of GC when all application threads are locked. It can be invoked with the **-Dibm.dg.trc.print=st_verify_heap** command-line option. It verifies the heap before the sweep phase and at the end of GC. **verifyHeap** walks the heap from the bottom to the top, until any of the following conditions occurs, or it reaches the end of the heap:

- The length of a chunk of memory is zero or too big to fit onto the heap.
- If the alloc bit is set (live object) and its method table or class block is NULL or invalid.
- If **verifyHeap** shows a problem before GC, that usually means that the problem was created by allocation routines or something outside the GC. If **verifyHeap** after GC shows a problem, while the **verifyHeap** before that GC has not shown any problems, it is likely that the problem has been created by GC.

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

When you see (from verboseGC) that the Java heap has a lot of free space, but the allocation request still fails, it usually points to a fragmented heap. To confirm this, run with **-Dibm.dg.trc.print=st_verify**. This option gives the

number of pinned and dosed objects, a high number of which indicates a fragmented heap. Running with **-Dibm.dg.trc.print=st_compact_verbose** lists the pinned and dosed objects.

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

You see this when GC decides to clear all soft references. **gc0()** is called once to do the regular garbage collection, and might run again one or two times to clear soft references. So you might see more than one GC cycle for one allocation failure.

# 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.

## Eager and lazy loading

The JVM must be able to load JVM class files. The JVM class loader loads referenced JVM classes that have not already been linked to the runtime system. Classes are loaded implicitly because:

- The initial class file - the class file containing the **public static void main(String args[])** method - must be loaded at startup.
- Depending on the class policy adopted by the JVM, classes referenced by this initial class can be loaded in either a lazy or eager manner.

An eager class loader loads all the classes comprising the application code at startup. Lazy class loaders wait until the first active use of a class before loading and linking its class file.

The first active use of a class occurs when one of the following occurs:

- An instance of that class is created
- An instance of one of its subclasses is initialized
- One of its static fields is initialized

Certain classes, such as **ClassNotFoundException**, are loaded implicitly by the JVM to support execution. You may also load classes explicitly using the **java.lang.Class.forName()** method in the Java API, or through the creation of a user class loader.

The IBM JVM's class resolution is lazy by default. Specifying the **-Dibm.cl.eagerresolution** command-line option turns on eager class resolution. Lazy class resolution improves startup time of JVMs. For example, the number of classes loaded in a basic Java test reduces from approximately 1500 to approximately 300 with lazy loading.

## 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:

```
Bootstrap
|
Extensions
|
Application
```

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

It is possible to 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 prevents the case where code from an untrusted source tries 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 a custom class loader?

The three main reasons for wanting to create a custom 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 a custom 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
      }
 }
```

## The Persistent Reusable JVM (z/OS only)

IBM has developed the new Persistent Reusable Java Virtual Machine technology, which is available on z/OS. The Persistent Reusable JVM allows the use of multiple JVMs that share classes, and for each of these to be reset, thereby distributing the cost of starting the JVM over multiple runs.

The Persistent Reusable JVM consists of a master JVM and several worker JVMs, that together make a JVMSet. The master JVM controls the JVMSet by providing a system heap that contains the core API, as loaded by the bootstrap class loader, and shareable classes. This system heap is available to all worker JVMs.

The Persistent Reusable JVM introduces two new class loaders: the Trusted Middleware Class loader (TMC) and the Shareable Application Class loader (SAC). Classes that are loaded by the TMC can operate without restrictions and persist over a JVM reset. Classes loaded by the SAC are not trusted and therefore have a set of restrictions placed on them that prevents the JVM from becoming unresettable. The class repositories that these class loaders use are specified by the launcher at startup. The class loader hierarchy therefore becomes:

```
Bootstrap ClassLoader
|
Extensions ClassLoader
|
Trusted Middleware ClassLoader (TMC)
|
Shareable Application ClassLoader (SAC)
|
Application ClassLoader
```

Because of the parent-delegation model, classes are loaded by the correct loader, provided that they are placing into the correct class repository as defined by the command-line options:

- `-Dibm.jvm.trusted.middleware.class.path=<path>` and
- `-Dibm.jvm.shareable.application.class.path=<path>`

## WebSphere 5.0 ClassLoader overview

There are three major classes of ClassLoaders in the WebSphere system:

**System classloader**
Provided by the JVM

**WebSphere Runtime classloaders**
Used to load the WebSphere runtime and some supporting libraries for application use

**Application classloaders**
Used to load the application artifacts (Web Modules, EJB modules, Utility jars)

Each class loader is a child of the class loader above it. The application class loaders are children of the WebSphere Runtime class loader, which is a child of the System class loader.

For more information about application class loader policies and modes, refer to the WebSphere Software Information Center or refer to the Information Center that is part of your WebSphere installation.

# Chapter 4. Understanding the JIT

The JIT is the just-in-time compiler. It is not actually part of the Java Virtual Machine (JVM) but is, nonetheless, an essential component of Java. This chapter summarizes the relationship between the JVM and the JIT, and gives a short description of JIT operation.

## JIT overview

Java is an interpreted language, so it has a WORA (Write Once Run Anywhere) capability. The Java compiler outputs strings of *bytecodes*. The JVM turns those bytecodes into something that will execute on the host platform. A JVM that is interpreting bytecodes cannot match the performance of a native application that consists of machine code that an appropriate native compiler has generated.

The JIT is therefore important. In theory, the JIT comes into use whenever a Java method is called, and it compiles the bytecodes of that method into native machine code, thereby compiling it "just in time" to execute. After a method is compiled, the JVM calls that method-compiled code directly instead of trying to interpret it.

However, when the JVM starts, thousands of methods are executed. A significant overhead exists on all of them because of the time it takes the JIT to run and compile them. So, if you run without a JIT, the JVM starts up quickly but runs slowly. If you run with a JIT, the JVM starts up slowly, then runs quickly. At some point, you might find that it takes longer to start the JVM than to run an application.

## MMI overview

The MMI, the JVM, and the JIT are tightly coupled. The MMI acts as a wrapper around all Java methods. Among other things, the wrapper determines where the method is, if it has been compiled, and maintains a JIT threshold count. The MMI interprets a method until its threshold count is reached. So high-use methods are compiled quite soon after the JVM has started; low-use methods are compiled much later or perhaps not at all. The effect of the MMI is therefore to spread the compilation of methods out over the life of the JVM. In this way, the JVM starts up quite quickly, but you do not lose performance benefit because methods become compiled when they reach the threshold. The threshold is carefully selected to get the maximum balance between startup times and runtime performance. Its value varies between 500 and 1000 according to the platform on which the JVM is running.

However, invoking a normal ″C-loop″ interpreter for those first 500+ times that a method is called is still too slow. The MMI uses a hand-crafted assembly code interpreter, which uses various techniques to increase performance.

It is possible to disable the MMI interpreter and go back to the traditional C-l, but this is not a runtime option. The JVM has to be recompiled to do this.

Finally, the MMI uses the native stack, where possible, instead of the Java stack, to save Java stack frames. The JIT is continually evolving. As optimization techniques are implemented, they open new possibilities of optimizing that are based on the code from the previous cycle.

# Runtime modes

Three different ways of running the IBM JVM are available:

1. Default. MMI and JIT both active
2. MMI off, JIT on
3. MMI off, JIT off

In case 2, the JVM is a pure 'JIT'd' system. All methods are compiled before being run for the first time.

In case 3, the JVM is a pure interpretive system. No code is compiled. Note that turning the JIT off automatically turns the MMI off also.

The MMI is an integral part of the JVM. The JVM, the MMI, and the JIT are tightly coupled in the IBM JVM. The JIT knows about JVM data structures and can insert data into the JVM.

# How the JIT optimizes code

When the JIT is called, it needs to understand the semantics and syntax of the bytecodes before it can compile the code correctly. This chapter does not contain much detail, but provides a summary of the phases of JIT analysis.

The compilation consists of four stages:

1. Bytecode optimization
2. Quad optimization
3. DAG optimization
4. Native code generation

The first three phases are mostly cross-platform code.

## Bytecode optimization

This is a relatively simple operation where a set of known optimizations are applied to the bytecodes. Optimizations include:

- Flow analysis
- Static method inlining
- Virtual method inlining
- Idiomatic translation
- Field privatization
- Stack and register analysis

Flow analysis might be applied more than once to take account of preceding optimizations. After optimization, the bytecodes are translated into 'quads', which can be regarded as a pseudo machine code. Optimization can now be applied in a way that is similar to the way that conventional native compilers use.

## Quad optimization

Quad optimization includes:

- Control flow optimization
- Data flow optimization
- Escape analysis
- Loop optimization

- Data flow analysis

Quad optimization typically requires repeated applications of these techniques.

### DAG optimization

A DAG (Direct Acyclic Graph) of the quads is generated and subjected to:

- Induction analysis
- Loop versioning
- Loop striding
- Induction removal
- Dead storage analysis

### Native code generation

Native code generation proceeds differently, depending on platform architecture, but is broadly split between Intel and Power-PC architectures. The compiled code is placed into the JVM process space and the MMI wrapper is changed to point to the compiled code. At any given time therefore, the JVM process consists of the executables and a set of JIT-compiled code that is dynamically linked to the MMI method wrappers that are in the JVM.

So, if you get a crash or a hang in code that is in the JVM process space yet outside the range of compiled code in that process, you have a problem with JIT'd code.

It is possible for the JIT, by placing hooks into the compiled code, to revisit compiled methods and to recompile them with reference to operational data.

## JIT frequently-asked questions

**Can I disable the JIT?**
Yes. Set the appropriate command line parameter (see Appendix G, "Command-line parameters," on page 487) or environment variable (see Appendix E, "Environment variables," on page 407). Alternatively, delete or rename the JIT DLL, which is located with the JVM executables and called jitc.dll

**Can I use another vendor's JIT?**
No

**Can the JIT 'decompile' methods?**
That is, can compiled code be canceled? No.

**Can I control the JIT compilation?**
Yes. See Chapter 30, "JIT diagnostics," on page 295. Advanced diagnostics are available to IBM engineers.

**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 dynamically control the JIT?**
No. You can set the JIT initial conditions only at JVM start-up time. The JIT can be started up only at the same time as the JVM.

**Do special service arrangements exist for the JIT?**
At this time, no. Report to Java service problems that you think are JIT-related.

# Chapter 5. Understanding the ORB

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



## 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.

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 is, but less scalable. Because it has to conform to the CORBA specification for interoperability, RMI-IIOP is a more complex protocol. The developing of 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 website: 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

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).

## 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.UnicatRemoteObject 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.

## Stub 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 44. (Use the **-keep** option with rmic if you want to keep the intermediate .java files).

*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) | | |

In 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) .....
   orb.run();
  }
  catch(Exception e) {}
 }
}
```

For RMI-IIOP:

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

   // Get reference to the root poa & activate the POAManager
```

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

   // Create servant and its tie
   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(IIOP) 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);
  ```

- 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 (ex. 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.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.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, 1 or 2) (default: 0)

  When a GIOP 1.2 Request/LocateRequest/Reply/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/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.

- **com.ibm.CORBA.MaxOpenConnections**: (range: 0 through 255) (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 255) (default: 100)

The ORB cleans up only connections that are not busy from the connection cache table, if the size is of the table is higher than the MinOpenConnections.

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

  If this property is set to true, no local 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.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.

Table 3 shows the Sun properties that are now deprecated and the IBM properties that have replaced them .

*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.ORBInitialHost | com.ibm.CORBA.BootstrapHost |
| org.omg.CORBA.ORBInitialPort | com.ibm.CORBA.BootstrapPort |
| org.omg.CORBA.ORBInitialServices | com.ibm.CORBA.InitialReferencesURL |

Note that none of these properties are OMG standard properties, despite their names.

# 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

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 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 3).

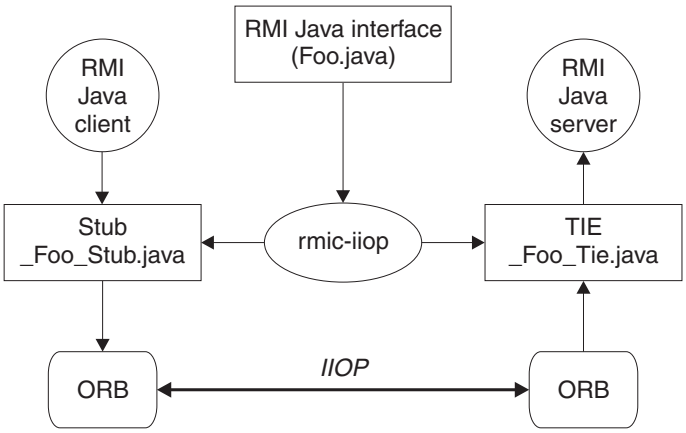



*Figure 3. The ORB client side*

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 constructor for that ORB class is called. This allows you to plug in a vendor ORB.

The ORB then 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:000000000000001d524d493a5......
```

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.

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 gets hold of 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, "Debugging the ORB," on page 187 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"
- "Tie generation"
- "Servant binding"
- "Processing a request" on page 56

### 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

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:

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

Figure 4 shows how the object adapter relates to the ORB, the skeleton, and the object implementation.

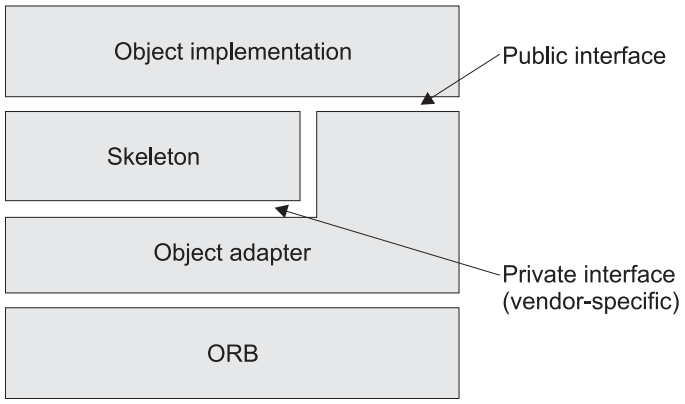



*Figure 4. Relationship between the ORB, the object adapter, the skeleton, and the object implementation*

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.

The IBM J2SE v.1.4 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 5 on page 59.

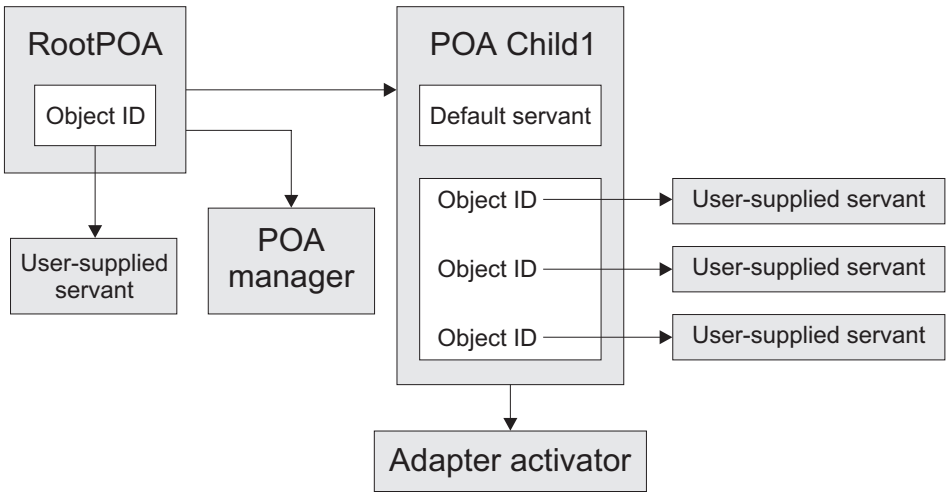



*Figure 5. Simple portable object adapter architecture*

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

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 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) {
               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 gets hold of 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:..."
```

### Other features

Among all the other differences with previous versions of IBM ORBs, it is important to outline the support for GIOP 1.2, an extended and improved RAS facility.

## IBM pluggable ORB

The IBM Java ORB is also made available for use with non-IBM J2SE implementations. This ORB is bundled with IBM middleware offerings, including the WebSphere Application Server and its various client packages. It can be used on platforms for which no IBM J2SE implementation is available, or where the customer has a business need to use an alternative J2SE implementation, but still requires the IBM ORB.

This release of the IBM Java ORB runs on the following SDKs:

- HP SDK for J2SEHP-UX 11i platform, adapted by IBM for IBM Software, Version 1.4.2
- HP Runtime Environment for J2SE HP-UX 11i platform, adapted by IBM for IBM Software, Version 1.4.2
- IBM 32-bit SDK for Solaris, Java 2 Technology Edition, Version 1.4.2
- IBM 32-bit Runtime Environment for Solaris, Java 2 Technology Edition, Version 1.4.2
- Sun Windows 32-bit SDK, v1.4.2

This version of the IBM Java ORB does not work with the IBM 32-bit SDK for Windows, Java 2 Technology Edition, Version 1.4.2.

The IBM Java ORB contains:

- ibmorbguide.htm
- ibm_bin directory
  - rmic - invoke rmic (HP-UX and Solaris)
  - idlj - invoke idlj (HP-UX and Solaris)
  - rmic.bat - invoke rmic (Windows)
  - idlj.bat - invoke idlj (Windows)
- ibm_lib directory
  - orb.idl - used by IDL compiler
  - ir.idl - used by IDL compiler
- jre\lib\endorsed directory
  - ibmext.jar - IBM JVM extended system emulation
  - ibmorb.jar - ORB runtime
  - ibmorbapi.jar - CORBA API
- lib directory
  - ibmtools.jar - rmic and idlj support

You must copy these files into the corresponding directories of the non-IBM SDK. For example, if the SDK is installed at /opt/j2sdk1.4.1, you must copy the runtime jar to /opt/j2sdk1.4.1/jre/lib/endorsed/ibmorb.jar.

## Using the IBM ORB runtime

The IBM Java ORB uses the "Java Endorsed Standards Override Mechanism" at http://java.sun.com/j2se/1.4.2/docs/guide/standards/index.html.

When you install the IBM Java ORB, the CORBA API provided in ibmorbapi.jar overrides automatically the CORBA API from your SDK. If you want to use your SDK's original CORBA API, move ibmorbapi.jar to another directory. If you then want to use the IBM CORBA API for a particular Java invocation, set the Java system property **java.endorsed.dirs** to include the directory to which ibmorbapi.jar has been moved, followed by the standard endorsed directory, <JAVA_HOME>\jre\lib\endorsed.

If you are using the CORBA API provided in ibmorbapi.jar, the IBM ORB runtime implementation is used by default.

If you are using your SDK's version of the CORBA API, you can set the IBM ORB runtime implementation to be the default with the following system properties:

```
org.omg.CORBA.ORBClass=com.ibm.CORBA.iiop.ORB

org.omg.CORBA.ORBSingletonClass=
          com.ibm.rmi.corba.ORBSingleton
javax.rmi.CORBA.UtilClass=
          com.ibm.CORBA.iiop.UtilDelegateImpl
javax.rmi.CORBA.StubClass=
          com.ibm.rmi.javax.rmi.CORBA.StubDelegateImpl
javax.rmi.CORBA.PortableRemoteObjectClass=
          com.ibm.CORBA.iiop.PortableRemoteObject
```

For instructions on how to set these properties, see http://java.sun.com/j2se/1.4/docs/api/org/omg/CORBA/ORB.html.

The tnameserv program, which is Sun's version of IBM name server, does not honor the orb.properties file. Therefore, if you use this program, the default SDK ORB is started. If you prefer to start the IBM name server when you are using the IBM ORB, you can start it with the command:

```
java com.ibm.CosNaming.TransientNameServer
```

## Using the IBM ORB development tools

The scripts ibm_bin\rmic.bat and ibm_bin\idlj.bat (on HP and Solaris, the files do not have the .bat extension) allow you to use the IBM version of **idlj** and the IBM back-end generators for the **rmic -iiop** and **-idl** options.

These scripts do not work on Windows 95 and Windows 98.

If you are invoking **idlj** from your application's code, you must invoke the main class com.ibm.idl.toJavaPortable.Compile.

If you want to invoke **rmic** from your application's code, ensure that lib\ibmtools.jar is on the application's classpath (not on the **rmic** classpath). The main class is sun.rmi.rmic.Main.

To check whether you are using the IBM Java ORB, use the **-version** option, which is valid only if used with **-iiop** or **-idl**:

# Chapter 6. Understanding the Java Native Interface

The Java Native Interface (JNI) is a source of much confusion. Much of this confusion occurs because the JNI specification, which is controlled by Sun Microsystems Inc, has not been fully understood. IBM strongly recommend that you read the JNI specification. Go to http://www.javasoft.com and search the site for JNI. Also search the site for educational information.

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 avoid the problems that can frequently occur in particular parts of JNI operation and design.

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*. It is a common mistake to write native JNI code that assumes implementation methods instead of conforming strictly to the specification. Although this code might not cause any problems at first, it could cause many problems if it is moved from one vendor's JVM to another, or if a vendor changes an implementation strategy.

The main topics that are discussed in the remainder of this chapter are:

# The JNI and the Garbage Collector

Before you read about the two main JNI topics ("Handling local references" on page 70 and "Handling global references" on page 72), you need to understand why and how references are maintained, and how the Garbage Collector is involved.

Three main interactions occur between the Garbage Collector and the JNI. Those interactions are:

1. Garbage Collector and object references
2. Garbage Collector and global references
3. Garbage Collector and retained garbage

The first two interactions manage Java objects in native code. The third is a result of the design of the IBM Garbage Collector.

## 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 JNIcode (...)
{
   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 JNIcode (...)
{
   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 `JNIcode` 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 70.

## Garbage Collector and global references

"Garbage Collector and object references" on page 68 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.

## Garbage Collector and retained garbage

Retained garbage is space that is unused in the heap, but not recognized as unused by the Garbage Collector. Therefore, the space is not reclaimed, it is retained. Retained garbage is garbage that might not be collected when you think it should be. For example, you know that a particular object is garbage but find that, after a garbage collection cycle, it has not been collected.

You cannot directly solve this problem; it usually solves itself. Eventually, the Garbage Collector finds the garbage. Do not assume that you can determine when garbage should be collected. If this simple answer is enough for you, go to "Handling local references" on page 70. Otherwise, continue here.

The retained garbage is a result of the conservative nature of the Garbage Collector reclamation and the use of JNI. You cannot always determine whether a value in the stack frame is a reference to a Java object, or whether it is a native parameter value that has been pushed onto the stack.

The Java threads execute as native threads on the native platform. The thread of execution is defined by the set of frames that is on the native stack. The Garbage Collector finds part of its set of root objects by scanning the native stack. When a mixture of native and Java frames exists on the stack, the Garbage Collector might scan native stack frames and create false root objects. These actions lead to retained garbage. The JVM attempts to store the limit of the heap when it changes from Java code to C/C++ code, so that it can control a garbage collection scan. However, nested or recursive JNI calls (for example, from native code -> Java -> native code -> Java) cause Java and native frames to become interleaved on the stack, and the Garbage Collector is forced to scan an area that does not contain valid heap references. As a result, false root objects are found, and the garbage of any object graph to which such a root object refers might be kept.

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

The IBM JVM usually uses a pinning implementation instead of a copy implementation.

## Handling local references

### Local reference scope

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 68 or have visited the Sun website at http://www.sun.com and read the documentation or specification that is given there.

It is very easy for a programmer to lose a local reference unintentionally. 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, the object is not pinned down, and 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 local reference scope as being the same as automatic variable scope. Local references go out of scope when the function they are "declared" in returns.

## 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

Occasionally, you might see a message such as:

```
"***ALERT: JNI local ref creation exceeded capacity
```

This message does ***not*** indicate an error. It is warning from the JVM that your application has more local references than can be contained in the storage that you first allocated for them. The local reference storage was described in the previous section. The message suggests that you might want to check your JNI code to see why you have many outstanding local references, and decide whether it would be better if you managed them yourself (see “Manually handling local references”). Normally, it is assumed that a function or method will not hold many references at the same time. If, however, you have designed you code to hold many references, you can ignore the message.

The JVM does ***not*** stop storing local references when this message appears; it extends the storage capacity, as necessary. The execution of your application is not affected in any way by this message, except for a small processing overhead. If your application is designed this way and the message becomes annoying, or if you are not willing to accept the overhead of recreating stack frames, JNI calls are available that enable you to increase the capacity of the local reference storage.

The JNI specification does not set the local reference capacity of a JVM, nor does it, require (or deny) use of this message. Therefore, this message might or might appear. If it does, it might appear at different times for different JVMs.

## 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 JVM has a fairly small limit, on the order of $10^5$. 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 JVM. This difference can lead you to think mistakenly that your application works on the vendor's JVM, but not on the IBM JVM.

## 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 70).

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.

The IBM JVM generally uses the pin implementation. 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").

**Attention:** isCopy flag summary:

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 72). 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 and free the space used by the copy.

**JNI_COMMIT**
Update the data on the Java heap and **do not** 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 prevent unnecessary copying and so on. If you do change the data, use 0, or JNI_COMMIT to ensure that your changes actually happen, or use JNI_ABORT if appropriate.

**Attention:** mode flag summary:

- 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/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 and so on. 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 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.

**Attention:** Synchronization summary:

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

If you think that you have a problem with the interaction between your native code and the JVM (that is, JNI problems), you can run diagnostics that help you check the JNI transitions. These diagnostics are all command line options and must be passed to the JVM at startup time. Because they are all extra command line options, they must be preceded by the an X (for extra); for example, -Xcheck:jni.

These are options that you might find useful:

- check:jni
- check:nabounds

### check:jni

This option causes a set of wrappers around the actual JNI functions to be activated. The wrappers perform checks on the incoming parameters such as:

- Check whether the call and the call that initialized JNI are on the same thread.
- Check whether the object parameters are valid objects.
- Check whether local or global references refer to valid objects.
- Check the type matching in get or set field operations.
- Check static and nonstatic field id validity.
- Check whether strings are valid and non-null.
- Check whether array elements are non-null.
- Match the types on array elements

This option is an expensive overhead, but it is quite thorough on input parameter validation.

### check:nabounds

This option works in the same way as check:jni does, and wraps some checks for array bounds around the JNI array functions.

# 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. |
| 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 74). | Memory leaks, heap fragmentation, or both. |
| When you update a Java object in native code, ensure synchronization of access. | Memory corruption. |

# Chapter 7. 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.

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.TCPTransport 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.

DGCClient implements the client side of the RMI Distributed Garbage Collection system. The external interface to DGCClient is the registerRefs() method. When a LiveRef to a remote object enters the JVM, it must be registered with the DGCClient to participate in distributed garbage collection. When the first LiveRef to a particular remote object is registered, a dirty call is made to the server-side 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 DGCClient periodically sends more dirty calls to renew its lease. The DGCClient tracks the local availability of registered LiveRef instances using phantom references. When the LiveRef instance for a particular remote object is garbage collected locally, a clean() call is made to the server-side 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 on the Sun Web site:

http://java.sun.com/j2se/1.4.2/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 the Sun Web site:

http://java.sun.com/j2se/1.4.2/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 Sun RMI FAQ Web site:

http://java.sun.com/j2se/1.4.2/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 8. 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. Java is vital to IBM's strategic products such as the IBM WebSphere Application Server.

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) and Bangalore (India). This geographical split is transparent to you for the purpose of submitting problem reports. However, you might find that you need to communicate directly with a service engineer, in which case be aware that Hursley operates on GMT and uses Daylight Savings Time (DST), while Bangalore operates on Indian Standard Time (IST), which is GMT + 4.5. India does not use 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.
- **Via 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 dependant 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 9. 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 86
- "Factors that affect JVM performance" on page 86
- "Test cases" on page 86
- "Performance problems — questions to ask" on page 86

See Part 3, "Problem determination," on page 95 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:

- Full version number
- Command line options
- Environment, non-default settings
- OS and OS version
- OS distribution (if applicable)
- Javadump
- Optionally, core dump (see Chapter 12, "First steps in problem determination," on page 97 for instructions on how to enable this)
- SDFF dump for use with the cross-platform dump formatter (see Chapter 14, "AIX problem determination," Chapter 15, "Linux problem determination," or Chapter 18, "Windows problem determination," as appropriate, for instructions about how to use the **jextract** command to create the SDFF dump from the core dump)
- SVC dump for z/OS; see Chapter 19, "z/OS problem determination."
- Optionally, cross-platform dump formatter (see Chapter 12, "First steps in problem determination," on page 97 for instructions on how to enable this)
- Heapdump, where required
- Verbose output, where required
- 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 95.

## Things to try

Refer to Chapter 12, "First steps in problem determination," on page 97.

## Factors that affect JVM performance

- Runtime flags
- Environment variables (list required environment variable)
- Set stack and heap size, Memory size (**MAXDATA** setting and **-Xms**, **-Xmx** , **-Xss**, and **-Xoss** settings)
- The search path to the class libraries (class path, mostly used classpath should come first)
- Garbage collection
- System limits
- The quality of the code
- System thread parameters
- The machine configuration
- I/O disk size and speed
- Number and speed of CPUs
- 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 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 10. 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

See "Submitting Java problem reports to IBM" on page 83.

## 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 11. Submitting data with a problem report

Having 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. This chapter describes how to transmit data to IBM Java service. Data can be sent to IBM in three ways:

- Java service maintain an anonymous ftp server, named 'javaserv', for sending or receiving data, This server is behind the IBM firewall and is therefore accessible only inside IBM. Ask your SE to transmit the data.
- IBM also maintains an anonymous ftp public server. Java service prefer the use of the javaserv ftp because the IBM server is not under the control of the IBM Java Technology Center.
- 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 93). 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:



## IBM internal only (javaserv)

ftp to javaserv like this:

```
ftp javaserv.hursley.ibm.com
```

1. Log-in anonymously.
2. Change to directory pmrs and create a directory called 12345 (assuming your PMR is 12345.xxx.xxx).
3. Change into 12345.
4. Set bin mode.
5. PUT your files.

Your output should look like this:

```
H:\crashes > ftp javaserv.hursley.ibm.com
Connected to fat.hursley.ibm.com.
220 fat.hursley.ibm.com FTP server (Version 4.1 Tue Sep 8 17:35:59 CDT 1998) ready.
User (fat.hursley.ibm.com:(none)): anonymous
331 Guest login ok, send ident as password.
Password:
230 Guest login ok, access restrictions apply.
ftp> cd pmrs
250 CWD command successful.
ftp> mkdir 12345
257 MKD command successful.
ftp> cd 12345
```

```
250 CWD command successful.
ftp> bin
200 Type set to I.
ftp> put mytestcase
```

## Sending files to IBM support

1. ftp to testcase.boulder.ibm.com
2. Change to <platform>/to ibm. For example:
   - For Windows and AIX platforms, change to aix/toibm
   - For Linux, change to linux/toibm
   - For s/390, change to s390/to ibm
3. Set binary mode.
4. PUT your file

Your output should look like this:

```
H:\website\IntelW32 > ftp testcase.boulder.ibm.com
Connected to testcase.boulder.ibm.com.
220 testcase.boulder.ibm.com FTP server (Version wu-2.6.1(1) Thu Aug 16 13:39:44
 MDT 2001) ready.
User (testcase.boulder.ibm.com:(none)): anonymous
331 Guest login ok, send your complete e-mail address as password.
Password: fred@bloggs.customer.com
230-Please read the file README
230- it was last modified on Wed Oct 31 08:42:25 2001 - 29 days ago
230-Please read the file README_PS.TXT
230- it was last modified on Wed Oct 31 08:42:11 2001 - 29 days ago
230 Guest login ok, access restrictions apply.
ftp> cd aix
250 CWD command successful.
ftp> cd toibm
250 CWD command successful.
ftp> bin
ftp> put myfile
```

Files are kept on the server for only a short time, so notify IBM support immediately after you have sent the files.

## Getting files from IBM support

You can get files from IBM support in two ways:

1.
   a. Point your browser to http://testcase.software.ibm.com
   b. Click the **TESTCASE SERVER**.
   c. Click the **<platform>/fromibm** icons. For example:
      - For Windows and AIX platforms, change to aix/fromibm
      - For Linux, change to linux/fromibm
      - For s/390, change to s390/fromibm
   d. Click on the file that you want.
2. ftp to the server as above, and GET the data.

Remember that the files are 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.

## Sending an AIX core file to IBM support

In general, it is difficult to correctly examine an AIX core file that is not in the environment in which it is run. This is because the core file does not include any of the libraries that were loaded by the process at the point of failure. For IBM support to be able to use fully the data that is in the core file, you must make the loaded libraries available also. For this purpose, a tool, called libsGrabber.sh, is available. When run against a core file, libsGrabber.sh generates a list of libraries that were loaded, and their locations. From this list, it creates a compressed file. that contains the libraries and a copy of the core file. This compressed file contains all the files that IBM support requires to analyze the core files on another machine.

## 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 95) 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

IBM produces Software Development Kits (SDK) and Runtime Environments (JRE) for a number of different platforms, including:

- Windows 32- and 64-bit
- AIX (Power PC) 32- and 64-bit
- z/OS (S390)
- Linux for Intel 32-bit and AMD64, Linux for PowerPC 32- and 64-bit, and Linux for zSeries 31-bit and 64-bit..

To use this part, go to the chapter that relates to your platform. A chapter covers both 32- and 64-bit versions of the JDK for that particular platform where applicable. 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 on which you have easiest access.

If you use the IBM WebSphere Application Server, the above guidance applies to you, but read Chapter 13, "Working in a WebSphere Application Server environment," on page 99 first, because the platform-specific chapters discuss subjects such as environment variables, and you will need the additional information that is given in the chapter for the WebSphere Application Server.

A couple of JVM issues do not fit neatly into the platform model, and these have their own chapters:

- Chapter 20, "Debugging the ORB," on page 187
- Chapter 21, "NLS problem determination," on page 201

If you have problems in these areas, check out the appropriate chapter in addition to general diagnostics about your platform.

The chapters in this part are:

# Chapter 12. 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. Depending on the platform, core dumps might not be enabled by default (see Chapter 27, "JVM dump initiation," on page 251 for details). To enable core dumps, set the environment variable **JAVA_DUMP_OPTS** to:

```
JAVA_DUMP_OPTS="ONERROR(JAVADUMP,SYSDUMP) ONEXCEPTION(JAVADUMP,SYSDUMP),
 ONDUMP(JAVADUMP)"
```

See Appendix E, "Environment variables," on page 407 for details on setting environment variables.

**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 13 through Chapter 19).

**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. (See Chapter 13, "Working in a WebSphere Application Server environment," on page 99.)

**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 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 30, "JIT diagnostics," on page 295.)

**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 13. Working in a WebSphere Application Server environment

The WebSphere Application Server depends on the JVM and ORB technology. Refer to Appendix A, “Compatibility tables,” on page 397 for WebSphere Application Server/JVM/ORB compatibility tables.

The IBM JVM version 1.4.2 ships with WebSphere Application Server version 5.1.1 Earlier versions of WebSphere Application Server shipped with earlier versions of the JVM.

WebSphere Application Server 5.1.1 ships with the IBM JVM on Windows, AIX, Intel Linux, PPC Linux, and z/OS Linux in the 32-bit versions.

WebSphere Application Server 5.1.1 also ships with Sun and HP JVMs on the relevant Solaris and HP platforms. In these cases, IBM ships a ″hybrid″ Java SDK comprising the vendor’s JVM, the IBM ORB, and additional IBM packages such as security.

For aspects of WebSphere Application Server JVM support (for example, information on how to set JVM runtime parameters or how to get heapdumps from the WebSphere environment) visit the WebSphere Application Server support and service site at http://www.ibm.com/software/webservers/appserv/was/support/. Click on the **Technotes** link and search for the topic that interests you to find relevant documents.

# Chapter 14. AIX problem determination

This chapter describes problem determination on AIX in:



If you are working in the alternative debug environment, see Appendix I, "Using the alternative JVM for Java debugging," on page 499.

## 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 (AIX 5.1 onwards) if the hardware is 64-bit. In that case, you have to enable a 64-bit application environment by: **Smitty -> System Environments -> Enable 64-bit Application Environment**.

Occasionally the configuration process does not work correctly, or the environment might become altered, affecting the operation of the JVM. In these conditions, you can make a number of 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 for the Version 1.4.2 Developer Kit is in `/usr/java142`. For developer kits packaged with other products, the installation directory might be different. In such a case, 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/java142/jre/bin:/usr/java142/bin` in your **PATH** environment variable . If it is not present, add it by using `export PATH=/usr/java142/jre/bin:/usr/java142/bin:$PATH`
3. Ensure that the **LANG** environment variable is set to a supported locale. You can find the language environment in use using `echo $LANG`, which should report one of the supported locales as documented in the *User Guide* shipped with the SDK.
4. Ensure that all the prerequisite AIX maintenance and APARs have been installed. The prerequisite APARs and filesets will have been checked during an install using `smitty` or `installp`. You can find the list of prerequisites in the *User Guide* that is shipped with the SDK. Use `lslpp -l` to find the list of current filesets. Use `instfix -i -k <apar number>` to test for the presence of an APAR and `instfix -i | grep _ML` to find the installed maintenance level.

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

## Enabling full AIX core files

The AIX core file is created in the current working directory for the Java process. The core file can often be truncated if it is full of core dumps that have not been enabled in the operating system settings.

Provided with AIX v5.2 and upwards, the **syscorepath** utility can be used to specify a single system-wide directory in which all core files of any processes are saved. The syntax for this command is: `syscorepath -p alternate_directory`.

To set the OS for full core dumps and files to `unlimited`:

1. Set the **ulimit** setting for core dumps to `unlimited`: `ulimit -c unlimited`.
2. Set the **ulimit** setting for core files to `unlimited`: `ulimit -f unlimited`.
3. Set Smit to use full core dumps either by starting **smit** and setting: **System Environments -> Change/Show Characteristics of Operating System -> Enable Full CORE** dump to "**TRUE**", or by using the command `chdev -l sys0 -a fullcore='true'` as root.
4. Ensure that the current working directory has enough disk space available to write the core file. You can redirect AIX core files to alternative locations using a symbolic link. To do this, you must create a link from the current working directory of the Java process to an alternative directory where there is a file called "core":

   ```
   ln -s <alternative directory path>/core <current working directory of Java process> /core
   ```

After a full core file has been generated and located, you must rename that file to prevent any other core file, that is generated in the same directory, from overwriting it.

# 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, the AIX 4.3.3 and 5.1 publications can be obtained from the IBM Web site (go to www.ibm.com/aix and follow the links). Of particular interest are:

- The AIX 5.1 Performance Management Guide (**AIX 5L Version 5.1 Books -> System Management Guides -> Performance Management Guide**)
- The AIX 4.3 Performance Management Guide (**AIX Version 4.3 Books -> System Management Guides -> Performance Management Guide**)
- The AIX Programmer's Guides - the AIX 4.3.3 or AIX 5.1 Reference Documentation.
- The Redbook: *"C and C++ Application Development on AIX" (SG24-5674)* available from: http://www.redbooks.ibm.com.

## Other sources of information for debugging

Other sources of information for debugging AIX problems are:

- These articles on developerWorks:
  - http://www-106.ibm.com/developerworks/eserver/library/es-javaonaix_core.html
  - http://www-106.ibm.com/developerworks/eserver/library/es-JavaOnAix_install.html
- A set of presentation slides about installing, configuring, and debugging the JVM. Very good on annotated screen shots that walk you through the various processes:

ftp://ausgsa.ibm.com/projects/l/13java/public/docs/Implementing_Java_on_AIX.prz

- AIX technical support:
  http://techlink.austin.ibm.com/cgi-bin/austext/megacgi

## Starting Javadumps in AIX

See Chapter 25, "Using Javadump," on page 219.

## Starting Heapdumps in AIX

See Chapter 26, "Using Heapdump," on page 245.

## Debugging memory leaks

The dbgmalloc library can be linked in to a customer native library to help identify native memory leaks. dbgmalloc must be linked in to the library before the C-runtime library, so that the standard memory routines can be overridden.

Note that dbgmalloc is meant for IBM use only.

The following options must be added to the `makeC++SharedLib_r` command before any others:

```
-L$SDK/jre/bin -ldbgmalloc
```

(The environment variable `$SDK` points to the Java SDK directory (for example, /opt/IBMJava2-142).

For more information about AIX memory, see "Understanding memory usage" on page 115.

## AIX debugging commands

### 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**.

## 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 Addr Range
2c7ea        3 work shmat/mmap            36678     0    0 36656   0..65513
3c80e        4 work shmat/mmap             7956     0    0  7956   0..65515
5cd36        5 work shmat/mmap             7946     0    0  7946   0..65517
14e04        6 work shmat/mmap             7151     0    0  7151   0..65519
7001c        d work shared library text    6781     0    0   736   0..65535
    0        0 work kernel seg             4218  1552  182  3602   0..22017 :
                                                                    65474..65535
6cb5a        7 work shmat/mmap             2157     0    0  2157   0..65461
48733        c work shmat/mmap             1244     0    0  1244   0..1243
 cac3        - pers /dev/hd2:176297        1159     0    -     -   0..1158
54bb5        - pers /dev/hd2:176307         473     0    -     -   0..472
78b9e        - pers /dev/hd2:176301         454     0    -     -   0..453
58bb6        - pers /dev/hd2:176308         254     0    -     -   0..253
 cee2        - work                         246    17    0   246   0..49746
4cbb3        - pers /dev/hd2:176305         226     0    -     -   0..225
7881e        - pers /dev/e2axa702-1:2048    186     0    -     -   0..1856
68f5b        - pers /dev/e2axa702-1:2048    185     0    -     -   0..1847
28b8a        - pers /dev/hd2:176299         119     0    -     -   0..118
108c4        - pers /dev/e2axa702-1:1843    109     0    -     -   0..1087
24b68        f work shared library data      97     0    0    78   0..1470
64bb9        - pers /dev/hd2:176311          93     0    -     -   0..92
74bbd        - pers /dev/hd2:176315          68     0    -     -   0..67
3082d        2 work process private          68     1    0    68   65287..65535
```

```
10bc4        - pers /dev/hd2:176322          63     0    -     -   0..62
50815        1 pers code,/dev/hd2:210969      9     0    -     -   0..8
44bb1        - pers /dev/hd2:176303           7     0    -     -   0..6
7c83e        - pers /dev/e2axa702-1:2048      4     0    -     -   0..300
34a6c        a mmap mapped to sid 44ab0       0     0    -     -
70b3d        8 mmap mapped to sid 1c866       0     0    -     -
5cb36        b mmap mapped to sid 7cb5e       0     0    -     -
58b37        9 mmap mapped to sid 1cb66       0     0    -     -
1c7c7        - pers /dev/hd2:243801           0     0    -     -
```

in which:

**Vsid**
Segment ID

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

**Type**
Identifies the type of the segment:

**pers** Indicates a persistent segment.

**work** Indicates a working segment.

**clnt** Indicates a client segment.

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

**Description**

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

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

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

**kernel**
The segment is used by the kernel.

**shared library**
The segment is used for shared library text or data.

**process private**
Private data for the process.

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

**Inuse**
The number of pages in real memory from this segment.

**Pin**
The number of pages pinned from this segment.

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

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

## 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.

## 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 200000000      Processor Speed False
```

## netpmon

This command uses the **trace** facility to obtain a detailed picture of network activity during a time interval. It also displays process CPU statistics that show:

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

For example,

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

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

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

```
Device                   Pkts/s  Bytes/s  Util  QLen   Pkts/s  Bytes/s   Demux
-------------------------------------------------------------------------------
token ring 0             288.95    22678  0.0%518.498   552.84    36761  0.0222
...
DEVICE: token ring 0
recv packets:           11074
  recv sizes (bytes):   avg 66.5    min 52      max 1514    sdev 15.1
  recv times (msec):    avg 0.008   min 0.005   max 0.029   sdev 0.001
  demux times (msec):   avg 0.040   min 0.009   max 0.650   sdev 0.028
xmit packets:           5788
  xmit sizes (bytes):   avg 78.5    min 62      max 1514    sdev 32.0
  xmit times (msec):    avg 1794.434 min 0.083   max 6443.266 sdev 2013.966
You can also request for less information to be gathered. For example to look at
 socket level traffic use the "-O so" option.

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

  netpmon -O so -t -o /tmp/netpmon_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
```

## 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 in AIX 4.3, 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` and get some back. 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.

## 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-1.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.

## 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
          1       35      42       0      22
          -       33      45       0      22

Average   0       32      47       0      20
          1       34      45       0      22
          -       33      46       0      21
```

## 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
Kernel   63.1   |#################          |  Reads         8  Rawin         0
User     36.8   |##########                 |  Writes     2469  Ttyout        0
Wait      0.0   |                           |  Forks         0  Igets         0
Idle      0.0   |                           |  Execs         0  Namei         4
                                               Runqueue   11.5  Dirblk        0
Network  KBPS   I-Pack  O-Pack   KB-In  KB-Out  Waitqueue   0.0
lo0     213.9   2154.2  2153.7   107.0   106.9
tr0      34.7     16.9    34.4     0.9    33.8  PAGING           MEMORY
                                               Faults     3862  Real,MB    1023
Disk    Busy%     KBPS     TPS KB-Read KB-Writ  Steals     1580  % Comp     27.0
hdisk0    0.0      0.0     0.0     0.0     0.0  PgspIn        0  % Noncomp  73.9
                                               PgspOut       0  % Client    0.5
Name         PID CPU% PgSp Owner               PageIn        0
java       16684 83.6 35.1 root                PageOut       0  PAGING SPACE
java       12192 12.7 86.2 root                Sios          0  Size,MB     512
lrud        1032  2.7  0.0 root                                 % Used      1.2
```

```
aixterm    19502  0.5  0.7 root               NFS (calls/sec)  % Free     98.7
topas       6908  0.5  0.8 root               ServerV2       0
ksh        18148  0.0  0.7 root               ClientV2       0   Press:
gil         1806  0.0  0.0 root               ServerV3       0   "h" for help
```

### trace

This command captures a sequential flow of time-stamped system events. The trace is a valuable tool for observing system and application execution. While many of the other tools provide 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. Two postprocessing tools that can extract information from the trace are utld (in AIX 4) are curt (in AIX 5). These tools provide statistics on CPU utilization and process and thread activity. The third postprocessing tool is splat, the Simple Performance Lock Analysis Tool. This tool is used to analyze lock activity in the AIX kernel and kernel extension 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.

# 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, two documents are vital to debugging the problem:

- The AIX core file. Enter the command `jextract <core file name>` to take the native core file as input and produce a file in SDFF format (the input format required by the cross platform dump formatter).
- The JVM-produced Javadump file.

## Interpreting the stack trace

If dbx or stackit produce no stack trace, the crash usually has two possible causes:

- A stack overflow of the native AIX stack.
- JIT compiled or MMI code is currently running.

A failing instruction reported by dbx or stackit 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) does not understand the structure of the JIT and MMI 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 30, "JIT diagnostics," on page 295). 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.

## Sending an AIX core file to IBM Support

See "Sending an AIX core file to IBM support" on page 93.

# 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 (LK)" on page 222.

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 103. For example:

```
$ ps -fp 30450
    UID   PID  PPID  C    STIME     TTY   TIME CMD
    root 30450 32332  2   15 May  pts/17  12:51 java ...
```

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

## AIX infinite loops

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, ask why the I/O is not occurring. Has the process at the other end of the I/O failed? Do any network problems exist?

## Investigating busy hangs in AIX

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)
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 being done by the running threads, 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.

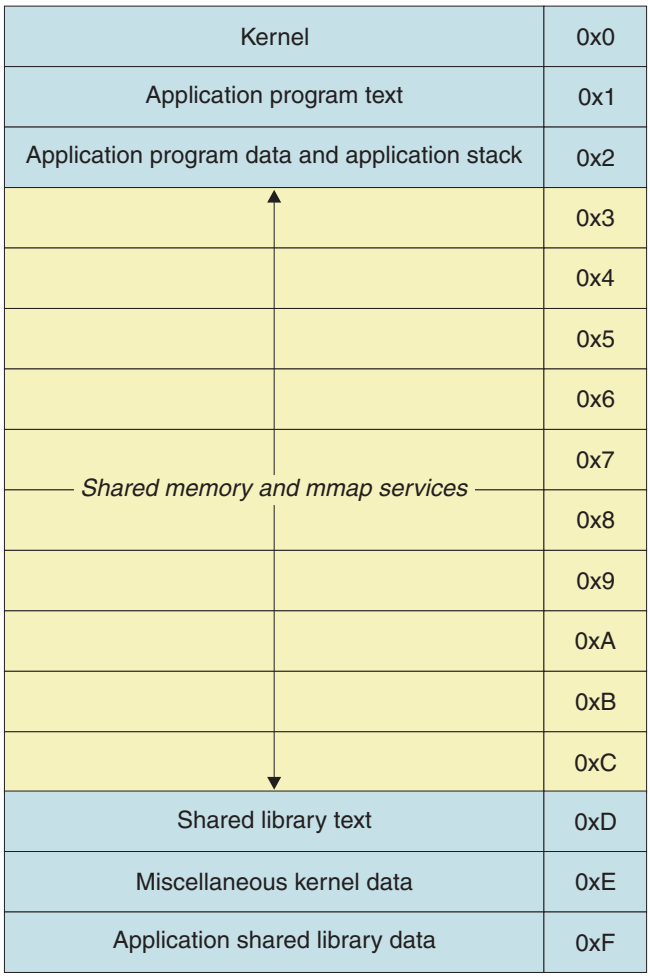



*Figure 6. The AIX 32–Bit Memory Model with MAXDATA=0 (default)*

- Segment 0 is assigned to the kernel.
- Segment 1 is application program data.
- Segment 2 is the 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)

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 `shmap()` data.

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.
- On later versions of AIX Version 4.3.3 and on AIX Version 5.1, setting the *o_maxdata* value by using the **LDR_CNTRL=MAXDATA=0xn0000000 (n segments)** environment variable.

Altering the MAXDATA applies only to a 32-bit process, and should not be done on the 64-bit JVM.

## The native and Java heaps

The JVM maintains two memory areas, the Java heap, and the native (or system) heap. These two heaps have different purposes, 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. In the AIX 1.2.2 JVM, this Java heap was allocated as one contiguous area of shared memory, running from the first available segment of shared memory up to the maximum heap size setting. Now, the Java heap is allocated using malloc, and therefore is placed at the next available area of process private memory. 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. Next, you can move the artificial heap size limit imposed by the minimum heap size setting 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 Java2 32-Bit JVM default memory models

In the AIX 1.2.2 JVM, the **MAXDATA** setting is set to 5 segments. This gives 5 segments for the native heap, and allows up to 5 segments to be used for the Java heap. (Theoretically there are 6 shared memory segments left, but because segment E is not contiguous to the rest of shared memory, it is not used.)

Now, the JVM has a **MAXDATA** setting of 8 segments. This is the maximum permissible value (segments B and C can be used only for shared memory). and provides a 2 GB limit for the combined Java and native heaps. Remember that the

Java heap is preallocated at the maximum heap size, so setting a large Java heap size reduces the amount of memory available to the native heap.

## Changing the memory models

You can change the memory model of the JVM in two ways:

1. Move from a malloc allocated heap to an **mmap** allocated heap.
2. Alter the **MAXDATA** setting.

You gain little by reducing the **MAXDATA** setting of the JVM while it is running with a malloc() allocated Java heap. In this case, lowering the **MAXDATA** setting reduces the available memory that is used by both the Java and native heaps.

To cause the JVM to use mmap instead of malloc to allocate the Java heap, set the environment variable: **IBM_JAVA_MMAP_JAVA_HEAP=true**, or alternatively, set a Java heap size of 1 GB or greater. A 1 GB heap causes the Java heap to be allocated from shared memory using mmap. If you do not change the **MAXDATA** setting from the default value of 8, only 2 contiguous segments of shared memory will be available for use by the Java heap, therefore imposing a maximum heap size of 512 MB.

After you have monitored the native heap usage, you can reduce the **MAXDATA** setting to allow greater Java heap sizes (at the cost of the native heap size).

## 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.

Under the 1.2.2 memory model, because the Java heap is allocated using mmap(), there can be no confusion whether memory allocated to a specific segment of memory (under 'Esid') is allocated to the Java or the native heap. With a **MAXDATA** setting of 5 segments, the primordial thread stack is held in segment 2, and the subsequent five segments are available for use by the native heap (segments 3 to 7).

The mmap() allocated Java heap then resides in the next segment, and occupies as many segments as it requires to allocate the maximum heap size as defined by the **-Xmx** command line value.

Here is the svmon output from the command that is shown above:

```
Pid Command Inuse Pin Pgsp Virtual 64-bit Mthrd
23,560 java 10,984 1,271 1,262 9,340 N Y

Vsid Esid Type Description  Inuse Pin Pgsp Virtual Addr Range
3b85 2 work process private  5,056 1 0 5,055 0..9499
                        65305..65535
b016 d work shared library text  2,090 0 24 581 0..65535
0 0 work kernel seg  1,651 1,257 1,238 3,481 0..21298 :
                        65475..65535
8c91 - pers /dev/hd2:153712  530 0 - - 0..2403
1,482 - pers /dev/hd2:22808  520 0 - - 0..842
......
```

In the 1.4 versions of the JVM, the Java heap is allocated using malloc(). This introduces some difficulties in interpreting whether memory is being allocated to the Java or native heap. As in the 1.2.2 memory settings, segment 2 is reserved for the primordial thread stack. The subsequent eight segments (from the **MAXDATA** setting of 8) are available for both the native and Java heaps. What usually occurs is that during startup of the JVM some addition threads and native objects will be allocated into segment 3. Next the Java heap will be allocated as a contiguous lump of memory of the size of the maximum heap size setting. Subsequent allocations to the native heap will occur after the Java heap.

Thus, you might to run with the 1.2.2 memory model if you suspect that you have a memory leak in the native heap as shown above.

## 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 environment variable `JITC_COMPILEOPT=COMPILING`. This prints each method name to stderr as it is being compiled and, as it finishes compiling, the location in memory where the compiled code is stored.

```
Compiling [java/io/BufferedOutputStream][flushBuffer]
345bafe0  [java/io/BufferedOutputStream][flushBuffer]
Compiling [java/io/OutputStream][flush]
345bb0c0  [java/io/OutputStream][flush]
Compiling [java/lang/String][indexOf]
345bb13c  [java/lang/String][indexOf]
Compiling [sun/io/CharToByteConverter][nextByteIndex]
345bb218  [sun/io/CharToByteConverter][nextByteIndex]
Compiling [java/io/Writer][write]
345bb2ec  [java/io/Writer][write]
Compiling [java/io/BufferedWriter][flushBuffer]
345bb438  [java/io/BufferedWriter][flushBuffer]
Compiling [java/io/OutputStreamWriter][flushBuffer]
345bb678  [java/io/OutputStreamWriter][flushBuffer]
```

If the Java heap is being allocated using malloc(), the location the compiled code is being written to is useful in determining the location of the Java heap.

When you have monitored how much native heap you are using, you can increase or decrease the maximum native heap available by altering the **MAXDATA** setting if the Java heap is allocated using mmap(). You should reduce this value only if you require the extra Java heap space this would allow.

Increase the native heap if you are failing to create new threads or you are receiving OutOfMemoryErrors that are not related to Java objects. Do this by increasing the **MAXDATA** value if the Java heap is allocated using mmap(), or by reducing the Java heap maximum size if it is allocated using malloc().

## 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 **-Xverbosegc** to cause a report to be written to stderr each time garbage collection occurs.

```
<AF[305]: Allocation Failure. need 528 bytes, 20915 ms since last AF>
<AF[305]: managing allocation failure, action=1 (0/63962104) (3145728/3145728)>
<GC: Fri Dec  7 11:52:40 2001
<GC(376): freed 45406840 bytes in 220 ms, 72> free (48552568/67107832)>
  <GC(376): mark: 204 ms, sweep: 16 ms, compact: 0 ms>
  <GC(376): refs: soft 0 (age >= 32), weak 12, final 219, phantom 0>
<AF[305]: completed in 223 ms>

<AF[306]: Allocation Failure. need 5576 bytes, 14693 ms since last AF>
<AF[306]: managing allocation failure, action=1 (404168/63962104) (3145728/3145728)>
<GC: Fri Dec  7 11:52:55 2001
<GC(377): freed 44582248 bytes in 218 ms, 71> free (48132144/67107832)>
  <GC(377): mark: 202 ms, sweep: 16 ms, compact: 0 ms>
  <GC(377): refs: soft 0 (age >= 32), weak 14, final 194, phantom 0>
<AF[306]: completed in 221 ms>

<AF[307]: Allocation Failure. need 15432 bytes, 140 ms since last AF>
<AF[307]: managing allocation failure, action=1 (11728960/63962104) (3145728/3145728)>
<GC: Fri Dec  7 11:52:55 2001
<GC(378): freed 33315928 bytes in 216 ms, 71> free (48190616/67107832)>
  <GC(378): mark: 200 ms, sweep: 16 ms, compact: 0 ms>
  <GC(378): refs: soft 0 (age >= 32), weak 0, final 0, phantom 0>
<AF[307]: completed in 219 ms>

<AF[308]: Allocation Failure. need 20168 bytes, 120 ms since last AF>
<AF[308]: managing allocation failure, action=1 (16384864/63962104) (3145728/3145728)>
<GC: Fri Dec  7 11:52:55 2001
<GC(379): freed 28654352 bytes in 211 ms, 71> free (48184944/67107832)>
  <GC(379): mark: 196 ms, sweep: 15 ms, compact: 0 ms>
  <GC(379): refs: soft 0 (age >= 32), weak 0, final 1, phantom 0>
<AF[308]: completed in 214 ms>
```

The verbose garbage collection output allows you to determine whether an `OutOfMemoryError` has been caused by the Java heap or the native heap. It also allows you to tune the size of the Java heap for the kind of performance you want for your application; you can have garbage collection running frequently, for short periods of time, or infrequently for longer periods.

## Receiving OutOfMemory errors

Any OutOfMemory condition that occurs could be due to either running out of Java heap or Native heap. 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 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 F, "Messages and codes," on page 415 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 **-Xverbosegc** 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 **-Xverbosegc** output by garbage collection occurring more and more frequently, with less memory being freed. Eventually the JVM will fail, and the 'totally out of heapspace' message can be seen. (See Chapter 2, "Understanding the Garbage Collector," on page 7 for more details on **-Xverbosegc** 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 26, "Using Heapdump," on page 245. Similar information can be gained by using other tools, such as JProbe and OptmizeIt.

If an allocation of hundreds of kilobytes is failing with plenty of free space available, some fragmentation might be occurring. (See "Avoiding fragmentation" on page 22 to help you solve the problem.) If your system is suffering from fragmentation, make sure that the minimum heap size is smaller than the maximum, and preferably as small as possible (32 to 64 MB). A small size forces the initial class allocations, threads, and persistent objects in the bottom of the thread stacks to be allocated at the bottom of the heap. By forcing them together, you might reduce fragmentation. A small size also allows you (at least initially) to expand the heap to allocate large objects when you cannot allocate inside the heap because of fragmentation.

## 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.

It is important to remember that Java is not the only component that might be allocating memory to the Java processes native heap. Any JNI that is running, either as part of the application, or through loaded third-party libraries, will also malloc() to the native heap. The dbgmalloc library can be linked in to a customer

native library (see “Debugging memory leaks” on page 103), but this cannot be done with third-party libraries, and so eliminating them from the scenario is the easiest way of determining where any leak might be.

Note that dbgmalloc is meant for IBM use only.

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 81, 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. **-Xverbosegc** 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

# Debugging performance problems

Performance problems are often difficult to diagnose and fix. Usually, a performance problem is seen as a slowing down of the entire system and not as the failure of a particular component. A user normally sees a performance problem as a lack of responsiveness from the system, but this lack of response can be caused by any one of a large number of systems that interact.

An example of this is a system that has many users logged in from remote terminals over a network with several routers. The users might report that the system is slow because they experience long delays between typing and seeing the characters on their terminals. This could either be caused by the server being overcommitted and suffering 100% CPU usage or by packets being lost over the network.

## Finding the bottleneck

Given that any performance problem could be caused by any one of several other problems, you must look at several areas, and eliminate each as a possibility. First, determine which resource is constraining the system:

- CPU
- Memory
- Input/Output (I/O)

To do this, use the **vmstat** command. The **vmstat** command produces a compact report that details the activity of these three areas:

```
> vmstat 1 10
```

outputs:

```
kthr     memory             page              faults        cpu
----- ----------- ------------------------ ------------ -----------
 r  b   avm   fre  re  pi  po  fr   sr  cy  in   sy  cs us sy id wa
 0  0 189898   612   0   0   0   3   11   0 178  606 424  6  1 92  1
 1  0 189898   611   0   1   0   0    0   0 114 4573 122 96  4  0  0
 1  0 189898   611   0   0   0   0    0   0 115  420 102 99  0  0  0
 1  0 189898   611   0   0   0   0    0   0 115  425  91 99  0  0  0
```

```
 1  0 189898   611   0   0   0   0    0   0 114  428  90 99  0  0  0
 1  0 189898   610   0   1   0   0    0   0 117  333 102 97  3  0  0
 1  0 189898   610   0   0   0   0    0   0 114  433  91 99  1  0  0
 1  0 189898   610   0   0   0   0    0   0 114  429  94 99  1  0  0
 1  0 189898   610   0   0   0   0    0   0 115  437  94 99  0  0  0
 1  0 189898   609   0   1   0   0    0   0 116  340  99 98  2  0  0
```

The example above shows a system that is CPU bound. This can be seen as the user (us) plus system (sy) CPU values either equal or are approaching 100. A system that is memory bound shows values of page in (pi) and page out (po) exceeding 10 pages per second. 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 under "AIX debugging commands" on page 103.

## 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:

```
Starting Trace now
Tue Nov 26 11:40:11 2002
System: AIX voodoo Node: 4 Machine: 00455F1B4C00

Starting  sleep 60
Trace is done now
 * Samples from __trc_rpt2
 * Reached second section of __trc_rpt2
> cat __prof.all
          Process      PID      TID    Total   Kernel     User   Shared    Other
          =======      ===      ===    =====   ======     ====   ======    =====
             java    38454    43763     6700        5        0        3     6692
                X    15552    19681       29        7       11       11        0
             java    38454    35017       27       15        0       12        0
          swapper        0        3       25       25        0        0        0
              gil     1032     2065       23       23        0        0        0
           dtterm    20640    28639       20       11        0        9        0
              gil     1032     1291       17       17        0        0        0
              gil     1032     1807       15       15        0        0        0
              gil     1032     1549       13       13        0        0        0
            dtpad    21166    23867        9        1        0        8        0
            syncd     3136     3911        3        3        0        0        0
               sh    19436    38455        3        3        0        0        0
            sleep    19436    38455        2        2        0        0        0
             init        1      261        1        1        0        0        0
        rpc.lockd    12194    13421        1        1        0        0        0
             dsmc    27484    22709        1        0        0        1        0
            trace    16352    32479        1        1        0        0        0
            tprof    36242    37939        1        0        1        0        0
             nfsd    10848    55077        1        1        0        0        0
          =======      ===      ===    =====   ======     ====   ======    =====
            Total                       6892      144       12       44     6692

      Segment ::  5  28741921144832

          Process     FREQ    Total   Kernel     User   Shared    Other
          =======      ===    =====   ======     ====   ======    =====
             java        2     6727       20        0       15     6692
              gil        4       68       68        0        0        0
                X        1       29        7       11       11        0
          swapper        1       25       25        0        0        0
           dtterm        1       20       11        0        9        0
            dtpad        1        9        1        0        8        0
```

```
      syncd        1        3        3        0        0        0
         sh        1        3        3        0        0        0
      sleep        1        2        2        0        0        0
       init        1        1        1        0        0        0
rpc.lockd        1        1        1        0        0        0
       dsmc        1        1        0        0        1        0
      trace        1        1        1        0        0        0
      tprof        1        1        0        1        0        0
       nfsd        1        1        1        0        0        0
    =======      ===    =====   ======     ====   ======    =====
      Total       19     6892      144       12       44     6692


 Total System Ticks: 6892 (used to calculate function level CPU)
```

This output shows that the Java process with Process ID (PID) 38454 is using the majority of the CPU time. Notice the *Kernel* and *Shared* values. Because these are both very small, you know that only a small number of system calls exist. The majority of time is spent in *Other*, which accounts for time spent running actual Java code, either through the Mixed Mode Interpreter (MMI) or the Just In Time (JIT) Compiler.

In this example the JVM is performing normally and is doing a large amount of work (although no output is seen). At this point, two options are available to improve the performance of the Java application:

- Use a faster processor, or
- Profile the Java application for improvements by using Java profiling tools; for example, Hprof and JinSight. (See Chapter 38, "Using third-party tools," on page 383.)

If the results show that the Java process is using the majority of the CPU, and most of the time is not in "Other" as below, it is likely to be a JVM internal problem.

```
Process      PID      TID    Total   Kernel     User   Shared    Other
=======      ===      ===    =====   ======     ====   ======    =====
   java   162918   861337     3139     2847        0      292        0
   wait      774      775     2597     2597        0        0        0
   wait     3354     3355     1713     1713        0        0        0
   wait     2838     2839     1692     1692        0        0        0
```

However, the JVM might be performing abnormally long garbage collection cycles, or very frequent garbage collection, therefore reducing actual application throughput. To find out if this is the case, you can either:

- Look at the shared library usage shown in the tprof output, or
- Monitor the verbose GC output. This is easier, however, if the JVM was not already running with this option. You will have to restart it.

The relevant section of the tprof output to look at is the shared library section:

```
Shared Object                          Ticks  %   Address    Bytes
=============                          ===== ==== ========== ======
/applications/speople/hr81705/jre/bin/libjitc.a/  2052  5.2 d34ea000   22da60
/usr/lib/libc.a/shr.o                     1355  3.4 d016ebe0   1d64f7
/home/oracle/product/8.1.7/lib/libclntsh.a/shr.o  1081  2.7 d2dd7100   55d5d9
/applications/speople/hr81705/jre/bin/classic/libjvm.a/  1053  2.7 d3350000   1990ca
/applications/speople/hr81705/bin/libpscompat.a/   565  1.4 d0d4b000   26102
/applications/speople/hr81705/bin/libpsbtunicode.a/   311  0.8 d2bbe000   100355
/applications/speople/hr81705/bin/libpsmgr.a/   255  0.6 d1975000   bb5a3
/applications/speople/hr81705/bin/libpssys.a/   247  0.6 d0ded000   7f9ea2
/applications/speople/hr81705/bin/libpsora.a/   192  0.5 d2dc7000   fe82
/applications/speople/hr81705/bin/libpspcm.a/   125  0.3 d1c6f000   5f089c
/applications/speople/hr81705/bin/libpscmn.a/    84  0.2 d0d72000   576e7
/usr/lib/libpthreads.a/shr_xpg5.o           75  0.2 d0004000   20307
```

This shows that two of the JVM's libraries in the top four utilized: libjitc.a, the JIT Compiler, and libjvm.a, the core JVM itself. Examine the most highly used methods in each:

```
 Profile: /applications/speople/hr81705/jre/bin/libjitc.a/
 Total Ticks For All Processes (/applications/speople/hr81705/jre/bin/libjitc.a/) = 2052
Subroutine               Ticks  %   Source     Address Bytes
=============            ===== ==== ========   ========== =====
.fi_init                   293 0.7 /userlvl/ca141/src/jit/pfm/ppc/aix/../../../pfm/ppc/rt_framec.c d34f4fb8   218
._fill                      81  0.2 noname   d34ecb24   70
.dopt_dessa_dag             67  0.2 /userlvl/ca141/src/jit/pfm/ppc/aix/../../../sov/Dopt/dopt_rename.c d36a258c   20ac
.MERGE_VARREF               57  0.1 /userlvl/ca141/src/jit/pfm/ppc/aix/../../../sov/util/jit_dataflow.c d35e0e9c   234
.Copypropa_Init_Dataflow    53  0.1 /userlvl/ca141/src/jit/pfm/ppc/aix/../../../sov/Qopt/dfQ_copypropa.c d35fcca0   37c8
.Commoning_Final_Dataflow_B    47  0.1 /userlvl/ca141/src/jit/pfm/ppc/aix/../../../sov/Qopt/dfQ_commoning_sub.c d3648350   408c
.hasher                     43  0.1 /userlvl/ca141/src/jit/pfm/ppc/aix/../../../sov/util/hasher.c d352b78c   1b0
.dopt_regenerate_dag        43  0.1 /userlvl/ca141/src/jit/pfm/ppc/aix/../../../sov/Dopt/dopt_dag.c d36914c0   408c
.dopt_generate_dag          33  0.1 /userlvl/ca141/src/jit/pfm/ppc/aix/../../../sov/Dopt/dopt_quad2dag.c d3683044   4788
.ReachDef_Q_Fwd_Visit_DataFlow_R    32  0.1 /userlvl/ca141/src/jit/pfm/ppc/aix/../../../sov/Qopt/dfQ_reachdef.c d36c5ed8   220

   Profile: /applications/speople/hr81705/jre/bin/classic/libjvm.a/
   Total Ticks For All Processes (/applications/speople/hr81705/jre/bin/classic/libjvm.a/) = 1053

Subroutine               Ticks  %   Source     Address Bytes
=============            ===== ==== ========   ========== ======
._fill                     428  1.1 noname   d3351098   88
.memcpy                    145  0.4 moveeq.s d3351700   1b8
.atomicSetTLHAllocbits      51  0.1 /userlvl/ca141/src/jvm/sov/st/msc/gc_alloc.c d3430f24   3a4
.localMark                  35  0.1 /userlvl/ca141/src/jvm/sov/st/msc/gc_mark.c d344fd60   1de4
.is_instance_of             21  0.1 /userlvl/ca141/src/jvm/sov/xe/common/jit.c d336941c   8c
.clProgramCounter2Method    19  0.0 /userlvl/ca141/src/jvm/sov/cl/clloadercache.c d349669c   23c
```

This output shows that the highest used JIT-support function is fi_init, and the highest used JVM function is _fill. With knowledge of the internals of the JVM, you can determine that these calls are used to create a Java stack trace. This shows a problem in the JVM itself, and should be reported with all available documentation. If the libjvm.a is the highest used JVM library, and the high-use methods in that library consist almost exclusively of localMark and gc0, the most likely cause is a garbage collection tuning problem. See Chapter 2, "Understanding the Garbage Collector," on page 7 for more information about tuning garbage collection.

## Memory bottlenecks

If the results of vmstat point to a memory bottleneck, you must find out which processes are using large amounts of memory, and which, if any, of these are growing. Use the **svmon** tool:

```
> svmon -P -t 5
```

This command outputs:

```
------------------------------------------------------------------------------
     Pid Command          Inuse      Pin     Pgsp  Virtual   64-bit    Mthrd
   38454 java             76454     1404   100413   144805        N        Y
------------------------------------------------------------------------------
     Pid Command          Inuse      Pin     Pgsp  Virtual   64-bit    Mthrd
   15552 X                14282     1407    17266    19810        N        N
------------------------------------------------------------------------------
     Pid Command          Inuse      Pin     Pgsp  Virtual   64-bit    Mthrd
   14762 dtwm              3991     1403     5054     7628        N        N
------------------------------------------------------------------------------
     Pid Command          Inuse      Pin     Pgsp  Virtual   64-bit    Mthrd
   15274 dtsessi           3956     1403     5056     7613        N        N
------------------------------------------------------------------------------
     Pid Command          Inuse      Pin     Pgsp  Virtual   64-bit    Mthrd
   21166 dtpad             3822     1403     4717     7460        N        N
```

This output shows that the highest memory user is Java, and that it is using 144805 pages of virtual memory (144805 * 4 KB = 565.64 MB). This is not an unreasonable amount of memory for a JVM with a large Java heap - in this case 512 MB.

If the system is memory-constrained with this level of load, the only remedies available are either to obtain more physical memory, or to attempt to tune the amount of paging space that is available by using the **vmtune** command to alter the **maxperm** and **minperm** values.

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

## I/O bottlenecks

This book does not discuss conditions in which the system is disk- or network-bound. For disk-bound conditions, use filemon to generate more details on which files and disks are in greatest use, and netstat to determine network traffic. A good resource for these kinds of problems is *Accelerating AIX* by Rudy Chukran (Addison Wesley, 1998).

## 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 101), 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 a symbolic link.

To obtain a core file, set `export DISABLE_JAVADUMP=TRUE`. If you run **java –fullversion** and the build date is later than January 2003, you must set the **IBM_NOSIGHANDLER** as well; for example: `export IBM_NOSIGHANDLER=TRUE`.

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, you should:

1. Rename the core file to prevent the core file from being overwritten by any subsequent core file.
2. Run the AIX dump extractor against the core file (type `jextract <core file name>`) to generate a cross platform dump format file (SDFF) that the dump formatter can work on. Note that this SDFF file will be a significant fraction of the size of the original AIX core file, so plenty of disk space is required.
3. Use the libsGrabber.sh tool, which is available from IBM support, to generate a compressed package that contains the core file and its associated libraries. This compressed file contains all the files that IBM support requires to analyze the core files on another machine.

   The libsGrabber.sh tool works only on core files created by a 32-bit executable. Alternatively, the snapcore utility is available from AIX 5.1

onwards. You can use snapcore to collect the same information. For example, `snapcore -d /tmp/savedir core.001 /usr/java142/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 (JVMDG304) 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:



If you are working in the alternative debug environment, see Appendix I, "Using the alternative JVM for Java debugging," on page 499.

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

### Working directory

The current working directory of the JVM process is where core files, Java dumps, heap dumps, and the JVM trace outputs, including Application Trace and Method trace, are outputted. Enough free disk space must be available for this directory. Also, the JVM must have write permission.

### Linux core files

A core file is an image of a process that is created by the operating system when the process terminates unexpectedly. This file can be very useful in determining what went wrong with a process. The production of core files can be enabled by default, depending on the distribution and version of Linux that you have.

Because truncated files are of no practical use, set the size of the Linux core file to "unlimited".

*Table 5. Usage of ulimit*

| Usage | Action |
|---|---|
| ulimit -c | # check the current corefile limit |
| ulimit -c 0 | # turn off corefiles |
| ulimit -c x | # set the maximum corefile size to x number of 1024-bytes |
| ulimit -c unlimited | # turn on corefiles with unlimited size |
| ulimit -n unlimited | # allows an unlimited number of open file descriptors |
| ulimit -p | # size of pipes |
| ulimit -s | # maximum native stack size for a process |
| ulimit -u | # number of user processes |
| help ulimit | #list of other options |

The core file is placed into the current working directory of the process, subject to write permissions for the JVM process and free disk space.

Depending on the kernel level, a useful kernel option is available that gives corefiles more meaningful names. As root user, the option `sysctl -w kernel.core_users_pid=1` ensures that core files have a name of the form ″Core.PID″.

## Threading libraries

Two different threading libraries are available for Linux. The IBM JVM supports both the more recent Native POSIX Threads Library for Linux (NPTL) and the Linuxthreads libraries. The Linuxthreads library is supported both with and without floating stacks. NPTL is available and is the default library for RedHat distributions since RHEL3 and for SuSE since SLES9.

If you suspect a problem in the threading area, you can try using Linuxthreads to see if the problem lies in NPTL.

To use Linuxthreads on RHEL3 and RHEL4, set the following environment variable:

```
export LD_ASSUME_KERNEL=2.4.19
```

To use Linuxthreads on SLES9, set:

```
export LD_ASSUME_KERNEL=2.4.21
```

## Floating stacks

On older distributions, only Linuxthreads are available. On the Intel 32-bit architecture, a problem exists in Linux kernels earlier than release 2.4.10. The operating system might lock up when running the IBM JVM with floating stacks enabled. The Java wrapper script detects the kernel version and sets an environment variable **LD_ASSUME_KERNEL=2.2.5**, which on RedHat systems loads the Linuxthreads library without floating stacks enabled.

If the JVM is loaded by the Invocation API, the **LD_ASSUME_KERNEL** must be set either on the command line or by the invoking program.

If floating stacks are not enabled, thread stacks are aligned on fixed boundaries. The **-Xss** flag that sets the size of a thread stack in the JVM has no effect.

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 25, "Using Javadump," on page 219.

## Starting heapdumps in Linux

See Chapter 26, "Using Heapdump," on page 245.

## Using the dump extractor on Linux

When a dump occurs, the structure and contents of the core file produced differ depending on platform. A cross-platform dump formatter can automate some of the tasks that are involved with studying a corefile. For the dump formatter to function, all corefiles must be converted to a common format. The Linux Dump Extractor converts a corefile obtained on a Linux machine to a corefile suitable for use by the dump formatter. To use the Linux Dump extractor, run the command:

```
jextract <corefile>
```

This command produces a modified core file with a .sdff file extension, which you might be asked to send to IBM service. See Chapter 29, "Using the dump formatter," on page 261 for details of the Cross Platform 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
- **-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>` will show 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 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_JAVA_DEBUG_PROG=gd
java
```

Then you receive a **gdb** prompt, and you supply the run command and the Java arguments:

```
r<java_argumnets>
```

**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 noprint**
**handle sigusr2 pass nostop noprint**

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 actual function name and the source file for the function. For example:

**#6 0x804c4d8 in myFunction () at myApplication.c**

To view more in-depth information about a function frame, use the **frame** command along with a parameter specifying the frame number. After you have selected a frame, you can display its variables using the command `print var`.

Use the **print** command to change the value of a variable; for example, `print var = newValue`.

The **info locals** command displays the values of all local variables in the selected function.

To follow the exact sequence of execution of your program, use the **step** and **next** commands. Both commands take an optional parameter specifying the number of lines to execute, 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 25, “Using Javadump,” on page 219). 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 29, “Using the dump formatter,” on page 261) 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 **JAVA_COMPILER=NONE**.
- JVM code.

Other tracing methods:

- **ltrace**
- **strace**
- **mtrace** - can be used to track memory calls and determine possible corruption
- RAS trace, described in Chapter 35, “Using the Reliability, Availability, and Serviceability interface,” on page 355.

## 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 **verbosegc** 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 (LK)” on page 222.

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 142).
- Java Dumps give monitor and lock information.
- verbosegc 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
- Java Process Examination Tool (procdata).

  When the Linux JVM hangs or loops, there are a number of things you can look at that can help determine the cause of the problem. Linux has a virtual filesystem (the /proc filesystem) that contains data about a particular process

including signal masks, allocated storage, and current instruction counter. This information must be written to a file before any intrusive debugging takes place (such as trying to take a javacore or attaching a debugger).

You must first find out the PID of the primordial thread and then run the procdata class on the PID; for example, `java com.ibm.jvm.linux.procdata 5886`.

You can discover the primordial thread by using the **-H** option of the **ps** command. This gives you a hierarchical list of processes and their PIDs with the primordial thread being the topmost Java thread in the hierarchy. For example, if the JVM has hung and you run `ps -H` from another session, you should get output like this:

```
ps -efH
peacocb   5884  5882  0 13:45 pts/3    00:00:00       /bin/ksh
peacocb   5886  5884  3 13:45 pts/3    00:00:02         java hwawt
peacocb   5887  5886  0 13:45 pts/3    00:00:00           java hwawt
peacocb   5888  5887  0 13:45 pts/3    00:00:00             java hwawt
peacocb   5889  5887  0 13:45 pts/3    00:00:00             java hwawt
peacocb   5890  5887  0 13:45 pts/3    00:00:00             java hwawt
peacocb   5891  5887  0 13:45 pts/3    00:00:00             java hwawt
peacocb   5892  5887  0 13:45 pts/3    00:00:00             java hwawt
peacocb   5893  5887  0 13:45 pts/3    00:00:00             java hwawt
peacocb   5894  5887  0 13:45 pts/3    00:00:00             java hwawt
peacocb   5895  5887  0 13:45 pts/3    00:00:00             java hwawt
peacocb   5896  5887  0 13:45 pts/3    00:00:00             java hwawt
peacocb   5897  5887  0 13:45 pts/3    00:00:00             java hwawt
```

In this example, 5886 is the PID of the primordial thread (5887 is the PID of the thread manager, and thus the parent PID of all the other threads in the JVM).

Running procdata against the PID of the primordial thread produces a file called PID.procdata (in this case 5886.procdata). This file contains the output of /proc/5886/maps in addition to /proc/pid/stat and /proc/pid/status for all the threads in the JVM. It then summarizes the data that is collected, and produces a table of threads and the signals that are blocked or pending. The typical signal masks encountered are identified and any masks that are unusual are flagged as such. Send this file to IBM Java Service as an aid to problem diagnosis.

This process is completely unintrusive of the failing JVM. Typically, you run this *before* the debugger (gdb) is attached to a failing JVM because the process of attaching gdb will change the signal masks. You should coordinate this process with other debugging techniques so that a consistent core file, javacore, and procdata output from a single failure are collected together.

## 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 dbgmalloc library can be linked in to a customer native library to help identify native memory leaks. dbgmalloc must be linked in to the library before the C-runtime library, so that the standard memory routines can be overridden.

The following options must be added to the gcc for the native library to wrap the memory access routines:

```
-Wl,--wrap -Wl,malloc -Wl,--wrap -Wl,calloc -Wl,--wrap -Wl,realloc -Wl,--wrap
-Wl,strdup -Wl,--wrap -Wl,strndup -Wl,--wrap -Wl,free
-L$SDK/jre/bin -ldbgmalloc
```

(The environment variable $SDK points to the Java SDK directory (for example, /opt/IBMJava2-142).

You can use the **backtrace** trace option to debug memory leaks. See the backtrace entry under "Detailed property descriptions" on page 326 for usage details.

The **mtrace** tool from GNU is also available for tracking memory calls. The Allocation Debugging section of http://www.gnu.org/manual/glibc-2.2.3/html_node/libc_toc.html specifies how to use this tool effectively. 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 26, "Using Heapdump," on page 245.

## Debugging performance problems

Locating the causes of poor performance is often difficult, because, although many factors can affect performance, the overall effect is often the same; that is, poor response or slow execution of your program.

Whether you want to find obvious performance bottlenecks, or tune general performance, find out as much as possible about your system and how it performs. Also, remember that when you correct one set of problems, you 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. So, to really improve performance, you must experiment by tuning different parameters, monitoring their effect, and retuning until you are satisfied that your system is performing acceptably.

### System performance

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. Some of the most-used **ulimit** parameters are discussed in Table 5 on page 130.

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 quite 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 132.

## 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, while 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 132.

## 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, **-Dibm.dg.trc.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 performance

In addition to looking at your overall hardware and system performance, you can tune several JVM parameters to further increase performance of your Java application. These parameters are normally set as Java command line options.

```
java [-options] class [args...]
```

OR

```
java -jar [-options] jarfile [args...]
```

where options include:

- **-Xgcpolicy:optavgpause**
- **-Xmx**
- **-Xms**
- **-Xgcpolicy:optthruput**

The Java heap size is one of the most important tunable parameters of your JVM. It is especially important if you are running several processes and JVMs on your system. The heap contains all Java objects (live and dead) and free memory.

Garbage collection is based on how full your heap is. Therefore, a large heap size delays the frequency of garbage collection, but when garbage collection does occur, it takes longer to complete.

What you consider to be an acceptable heap size depends on your application; you will certainly need to experiment. In addition to balancing the frequency and length of garbage collections, you must also remember that memory that is allocated to one applications heap is not available to other applications. This is an example of fixing a bottleneck in one area, by increasing heap size to decrease frequency of garbage collection, and causing problems somewhere else. For example, other processes might have to use paging to supplement their diminished memory. Under no circumstances should heap size be larger than physical memory.

**-Xms** sets the initial heap size while **-Xmx** sets the maximum heap size.

After you have set the heap size, the **verbosegc** command shows you information about garbage collection. The default garbage collection policy is **optthruput**, which generally gives the fastest throughput. However, by specifying **optavgpause**, you can help programs that are displaying erratic response times, although throughput will be slower. See Chapter 31, “Garbage Collector diagnostics,” on page 299 for more information.

### JIT

The JIT is another area that can affect the performance of your program. When deciding whether to use JIT compilation, you must make a balance between faster execution and increased compilation overhead. The JIT is on by default; you can turn it off by using one of the following Java options:

```
-Djava.compiler=NONE
```

or

```
-Djava.compiler=" "
```

It is useful to investigate the JIT when you are debugging performance problems. For more details about the JIT, see Chapter 4, "Understanding the JIT," on page 37 and Chapter 30, "JIT diagnostics," on page 295.

You can learn much about your Java application by using **hprof**, the nonstandard profiling agent. Statistics about CPU and memory usage are presented along with many other options. The hprof tool is discussed in detail in Chapter 36, "Using the JVMPI," on page 369. **-Xrunhprof:help** gives you a list of suboptions that you can use with hprof.

## 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 sdff file is useful for service (see "jextract" on page 262).

### 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 25, "Using Javadump," on page 219.

### 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 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.tar.gz:

```
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 133.

# Known limitations on Linux

## Threads as processes

The JVM for Linux implements Java threads as native threads. This 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 libpthreads.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** environment variable 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 130.)

## 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 2.1 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:

```
/usr/sbin/chkfontpath --add /opt/IBMJava2-131/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, particularly with regard to Japanese fonts, see the *User Guide* for your SDK.

## CORBA limitations

Bidirectional GIOP is not supported.

When running with a Java 2 SecurityManager, invocation of some methods in the CORBA API classes might cause permission checks to be made that could result in a SecurityExecption. 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 |

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

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

## Scheduler limitation on SLES 8

After SDK 1.4.1 first became available, applications that run on symmetric multiprocessor machines have been found to perform better with a new IBM Virtual Machine implementation of Java monitors. Because this new implementation either improves, or has no effect on, the performance of most applications, it is now the default behavior. However, on SLES 8, a few heavily-multithreaded applications might not perform so well with the new default. You can restore the old algorithm by setting the environment variable:

```
export IBM_JVM_MONITOR_OLD=<any value>
```

This problem occurs only with the Linux scheduler implementation on SLES 8. It does not occur in SLES 8 Service Pack 2 when the following kernel configuration is applied:

```
/sbin/sysctl -w kernel.sched_yield_scale=1
```

Neither does it occur with the default configuration of SLES 8 Service Pack 3.

# Chapter 16. 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 some IBM add-ons, such as security packages. 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 81 and submit a bug report.

# Chapter 17. 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 some IBM add-ons, such as security packages. 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 81 and submit a bug report.

# Chapter 18. Windows problem determination

This chapter describes problem determination on Windows in:



If you are working in the alternative debug environment, see Appendix I, "Using the alternative JVM for Java debugging," on page 499.

## Setting up and checking your Windows environment

The installation process of the SDK or JRE sets up everything for you. The installer uses the Windows InstallShield software. If you are using an IBM product with embedded Java (for example, WebSphere Application Server or WebSphere MQSI), the product installation process installs Java for you.

The install process is the same on all versions of Windows. These versions are supported :

- Windows 98
- Windows NT4
- Windows 2000
- Windows ME
- Windows XP

If you experience any difficulty after the installation:

- If you installed Java as part of an IBM product, refer to the manuals for that product.
- If you installed Java as a standalone product or if you manually installed Java, check the following environment variables.

  **PATH**
  The **PATH** variable must point to the directory of your Java installation that contains the file jre.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" on page 151. The ReportEnv tool is available on request from jvmcookbook@uk.ibm.com.

Figure 7 shows the ReportEnv tool.

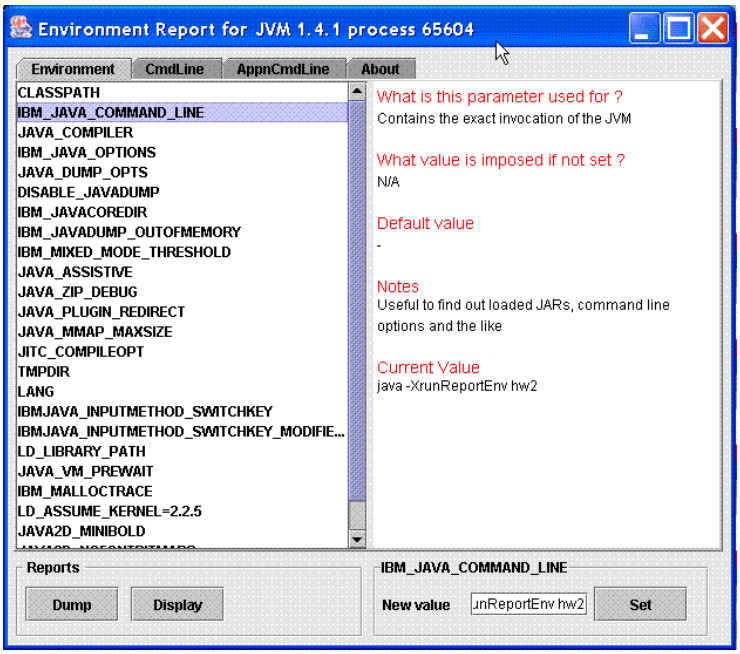



*Figure 7. Screenshot of the ReportEnv tool*

## Windows 32-bit large address aware support

From Version 1.4.2 Service Refresh 4 the IBM JVM for Windows 32-bit JVM includes support for the /LARGEADDRESSAWARE switch, also know as the /3GB switch. This switch increases the amount of space available to a process, from 2 GB to 3 GB. The switch is a Windows boot parameter, not a command line-option to the JVM.

This switch is useful in the following situations:

- Your application requires a very large number of threads.
- Your application requires a large amount of native memory.
- Your application has a very large codebase, causing large amounts of JIT compiled code.

To enable large address support, modify your boot.ini file and reboot your computer. For Instructions on how to do this, see the Microsoft website:

http://www.microsoft.com/whdc/system/platform/server/PAE/PAEmem.mspx. For a list of supported platforms, see this Microsoft knowledge base page: http://support.microsoft.com/kb/291988/.

After enabling the /3GB switch, the JVM gains 1 GB of extra memory space. This extra space does not increase the theoretical maximum size of the Java heap, but does allow the Java heap to grow closer to its theoretical maximum size (2 GB - 1 bytes), because the extra memory can be used for the native heap.

# 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 29, "Using the dump formatter," on page 261.

## Setting up for Javadump and Heapdump

Refer to Chapter 25, "Using Javadump," on page 219 and Chapter 26, "Using Heapdump," on page 245.

## Native Windows tools

Windows has embedded function for collecting data from processes that crash. The functionality is accessed from a utility that is called Dr. Watson. Logs from Dr. Watson and user dumps are useful to determine problems in crash and hang situations. Your Windows machine must be set up for collecting this data.

**Setting up Dr. Watson:** If Dr. Watson is set as your default debugger, a Dr. Watson log is generated whenever any process crashes. To set Dr. Watson as the default debugger, at a command prompt type `drwtsn32 -i`.

This installs Dr. Watson as the default application debugger. This is a one-time-only operation.

A Dr. Watson log is called drwtsn32.log by default.

**Setting up for a crash dump:** Dr. Watson provides an option that generates a crash dump file when a process crashes. To enable this:

1. Run drwtsn32 in a command prompt to get a Dr. Watson window.
2. Ensure that the checkboxes **Dump All Thread Contexts**, **Append To Existing Log File**, **Visual Notification**, and **Create Crash Dump File** are checked.
3. Ensure that the Dr. Watson log file and the Crash Dump files are stored in the directory that is indicated by the text boxes marked 'Log File Path' and 'Crash Dump' respectively. Set these paths to the appropriate directories.
4. Click **OK**.

Note that crash dumps are a complete dump of your computer virtual memory and can therefore be quite large. For example, if you have 4 GB of memory on your server, the crash dump size will also be in the GB range.

By default, crash dumps are put into a file that is called user.dmp and are called "user dumps" for this reason. Analyze user dumps with the **windbg** application that is provided by Microsoft®. You can also use a user dump to create a "minidump" that can subsequently be loaded into the IBM cross-platform debugger.

**Note:** By default, the Dr. Watson logs and crash dumps are put into your Windows installation directory. By default this directory is public, which

means that anyone on your network can access a crash dump. If you do not want this (for example, if the crash dump contains sensitive data such as passwords), use the Dr. Watson window to ensure these dumps are put somewhere private on your workstation.

**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.

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 160.

# 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 25, "Using Javadump," on page 219.

## Starting Heapdumps in Windows

See Chapter 26, "Using Heapdump," on page 245.

## Using the Windows Dump Extractor

The IBM Java Cross-Platform 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 29, "Using the dump formatter," on page 261 for details.

## Microsoft tools

Microsoft tools are provided as part of the operating system. It is therefore not appropriate for this book to provide detailed instructions on how to use these tools. Refer to www.microsoft.com and search on the site for instructions.

This section briefly describes how you might like to use the following tools:

- Dr. Watson
- User dumps
- WinDbg

### Dr. Watson

Dr. Watson is a post-mortem tool that you can configure to dump a Dr. Watson log whenever a Windows process crashes. The information that is in the Dr. Watson log can be useful. Refer to the manufacturer's instructions on how to enable the tool. When enabled, the tool always captures crash dumps until you disable it. However, with Javadump enabled, you cannot get Dr. Watson logs. You can overcome this limitation by passing the **-Xnosigcatch** option to JVM while invoking the Java application.

### User dumps

A user dump can be either a dump of a process, or a dump of the whole system. To get a user dump, you must configure Dr. Watson suitably.

Note that the data that is generated for a process dump is the same as the data that the dump extractor provides (see Chapter 29, "Using the dump formatter," on page 261). The dump extractor generates minidumps with associated process memory.

User dumps of a whole system are large; they consist of all the memory that is in your computer, including the swap file. Gigabyte dumps are quite common.

By default, user dumps are placed into a Windows directory that Windows makes shareable. You might want to keep user dumps more private if you are concerned about passwords and other security details that will be contained in a full user dump.

### WinDbg

WinDbg is the general Windows debugging tool. You can attach it to a running process, or use it in post-mortem mode by loading into WinDbg the user dump that is generated by the system.

## 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 JVMPI 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 25, "Using Javadump," on page 219 for more details on using Javadump.
- Collect the Dr. Watson log. Take a copy of the Dr. Watson log. See "Native Windows tools" on page 153 for details.

- Collect the crash dump. See “Setting up and checking your Windows environment” on page 151 for details.
- Run with the JIT turned off. See Chapter 30, “JIT diagnostics,” on page 295 for details.
- Collect the Javadump and the Dr. Watson logs 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 30, “JIT diagnostics,” on page 295.
- Try to disable the MMI. If the problem occurs with or without the JIT, try disabling the MMI. The MMI is the Mixed Mode Interface. MMI is switched on by default and delays compiling methods with the JIT until a certain threshold has been reached. This way, the JVM starts up reasonably quickly (no overhead of ″JITting″ all the basic methods) while retaining the advantages of having a JIT. However, with MMI active, some methods in your code are interpreted and some are executed as native code, depending on whether they have hit the MMI threshold. Set the MMI threshold to 0 to enforce ″JITting″ of all methods (that is, no code is interpreted). To do this, set the environment variable `IBM_MIXED_MODE_THRESHOLD = 0`. Run your application again and collect the Javadump and Dr. Watson logs. This is the opposite of the previous scenario where no code was ″JIT’d″.
- 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. As a quick check, run with the following parameters: `-nocompactgc -noclassgc -verbosegc`.
- 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 Dr. Watson and crash dumps.

## Tracing back from JIT’d code

You might get a crash in JIT’d code. If this happens, it is difficult to determine exactly what Java code is being executed.

### Identifying JIT’d code

The JIT compiles a Java method, then places the JIT’d code inside the Java process space. Two methods are available that you can use to identify the JIT’d code:

- Map file
- Process Explorer

**Using the map file:** The map file, which is generated when a Windows JVM is built, lists the components of the JVM (the DLLs), the load addresses, and the sizes of these. Here is a sample map:

```
jvm
Timestamp is 3e37b21c (Wed Jan 29 10:51:08 2003)

Preferred load address is 10000000

Start         Length     Name                   Class
0001:00000000 000fb2a0H .text                  CODE
0002:00000000 00000140H .idata$5               DATA
```

From this map, you can see that the preferred load address of the JVM dynamic link library, jvm.dll, is `0x10000000` and that the `CODE` segment is `000fb2a0H` bytes long.

When you get a crash, examine the Dr. Watson log, or disassemble the code in the dump formatter or WinDbg, at the point of the crash. If the code address is in the segment that contains the dll code (that is, address 0x1nnnnnnn), but is at an offset that is higher than 0xfb2a0, the code is JIT'd code.

**Using Process Explorer:** Process Explorer is described in "Process Explorer" on page 392. Look at the java.exe process and display the DLLs. Part of the display looks something like this:

```
- 0x10000000  0x155000  131.2003.0001.0023  29/01/03  <path>\classic\jvm.dll
```

In this case, the jvm dll is `0x155000` bytes long, so any code that you see above address 0X10155000 must be JIT'd code.

## Analyzing the dump

This section assumes that you are reasonably familiar with WinDbg. (Alternatively, you can use the command line debugger if you want.)

If the crash is in JIT'd code and if you have a Dr. Watson log and a user dump, you can manually trace back to find the method that was compiled into the JIT'd code. Note that the dump extract that is generated by the JVM is also compatible with the debugger.

1. Find the return address in the stack.
2. Find the end of the JIT frame.
3. Find the method name.
4. Find the class name, if needed.
5. Find the method signature, if needed.

## Finding the return address in the stack

1. From the Dr. Watson log, locate the section that is headed "State dump for current thread".
2. From the register dump, make a note of the stack pointer setting; that is, the value of the esp register.

This is all the information that you can get from the Dr. Watson log. Now, you must use WinDbg:

1. Start WinDbg.
2. Load the user dump: **File->Open Crash Dump**, load the user.dmp file.
3. Start debugging. Click **View->Memory** and enter the value of esp.

**Note:** By default, WinDbg dumps an address followed by the 16 words of data at that address and following:

```
d <address>
<address>             16 words
<address> + 16>  16 words
<address> + 32>  16 words
and so on
```

Because this dump cannot be shown on the printed page, in the following example, the addresses are shown in bold, and the words of data are allowed to wrap. Here is an example of the stack dump:

```
d 1650f2ac
0x1650f2ac 13245ca0 013aeae0 005ff620 005ff620 00000000 013aebe0 11cee0c9 11cedbec
 013aebe0 013aec00 005ff620 1650f314 00000001 12e40378 1650f2f0 71325739
0x1650f2ec 005ff620 01f799e0 00000000 00000000 18f2f258 18f30031 18f2dab0 00000000
 00000000 00000002 1650f34c 71325739 00000000 005ff620 00597660 01f799e0
```

The problem is that the stack frame is undefined, so the location of the return address in the frame is unknown. The only solution is to laboriously disassemble each address in the stack in turn. Eventually, you will see a disassembly that has a call instruction immediately in front of the disassembled instruction.

In the above example, disassembly showed nothing until selection of `11cee0c9` (the seventh word in the stack trace).

1. Click **View --> Disassembly**.
2. Click **Edit --> Go to Address**.
3. Enter the value `11cee0c9`. The disassembly view is something like this:

```
11CEE0C2 8BD5            mov         edx,ebp
11CEE0C4 E8E7AA0100      call        11D08BB0
11CEE0C9 0BD8            or          ebx,eax
11CEE0CB 8BC6            mov         eax,esi
```

Note that address `11cee0c9` disassembles to `or ebx,eax` and the immediately preceding instruction is `call 11d08bb0`. Therefore, `11cee0c9` is the return address in the JIT code.

You must now find the end of the JIT code section. The JIT aligns its code frames on a word boundary, so to find the end of the frame, you must start searching from `11cee0c8`, which is the return address rounded down to a word boundary.

## Finding the end of the JIT frame

The end of a JIT frame is flagged by a word that is set to `0xcccccccc`. To find this, dump out the frame from `11cee08` onward. The dump looks like this:

```
0x11CEE0C8 8bd80b00 8b088bc6 68528b11 0f013a83 0001038f 047a8100 004c73b0 0077850f
 528b0000 12b70f08 0491548b 854452ff 8bd58bc0 6d850fce 89ffffff 8b08245c
0x11CEE108 8b082444 8b10246c 8b14245c 8b182474 831c247c 89c320c4 eb08245c b9118be2
 02170fd2 e801ca83 5ed18e4c 72b9118b 83004cea 3de801ca 8b5ed18e 73b2b911
0x11CEE148 ca83004c 8e2ee801 178b5ed1 4c73b2b9 01ca8300 d18e1fe8 b9118b5e 004c73b2
 e801ca83 5ed18e10 ba515250 12fb7668 f2ff9bb9 803ee818 5a595ed1 e958d08b
0x11CEE188 fffffe73 ba515250 10cfb568 f2ffa0b9 8022e818 5a595ed1 e958d08b fffffe82
 ba515250 11441c58 f2ffbeb9 8006e818 5a595ed1 e958d08b fffffe94 ba515250
0x11CEE1C8 11441cb8 f2ffaab9 7feae818 5a595ed1 e958c88b fffffebe ba515250 11441c58
 f2ffbeb9 7fcee818 5a595ed1 e958d08b fffffef8 f2f19868 7f9ae918 f9835ed1
0x11CEE208 a7850f10 8bfffffe 884c8b08 fe9ce908 0489ffff 548d5124 5552ec24 8ffb386
8 31e85001 835ec2cb 855910c4 ba0f74c0 11cee0b8 048b0a89 fe7ce924 75e8ffff
0x11CEE248 905ed192 cccccccc 18f2f138 00000000 11cedf70 00000021 11cee2a8 11cee268
 00000001 11cee0c5 00000133 00000000 00000000 12e683c8 11cee258 00000019
```

The end of frame flag is at `11cee24c`. The next word (`18f2f138`) is the key, because it is a pointer to the method block (mb) that is inside the JVM. You do not need to understand method blocks except for the key offsets that are in it. The following offsets are valid:

mb + 0 pointer to classblock
mb + 4 pointer to method signature
mb + 8 pointer to method name string

When the classblock (cb) address is obtained, the following offset is needed:

cb + c pointer to class name string

### Finding the method name

Dump out memory at address 18f2f138 + 8. The dump looks like this:

```
d 18f2f140
0x18F2F140 12f3fdc8 0000401a 00000000 00000000 18f2ff94 00000000 18f30900
18f30ece 00000030 00000007 00000005 11cedf88 00030002 00000005 00000000
11cedfb0
```

Go to the memory view and dump from 12f3fdc8 with the display format as ASCII. This action displays the function name. In this case, it was removeItemChargeAdjustments().

Now you know that the JIT'd code is for the method removeItemChargeAdjustments. If this code is unique, you do not need to find the class name. Otherwise, you must find also the class name.

### Finding the class name

Dump the mb memory. The dump looks like this:

```
d 18f2f138 0x18F2F138 00650fd8 12f13518 12f3fdc8 0000401a 00000000 00000000
18f2ff94 00000000 18f30900 18f30ece 00000030 00000007 00000005 11cedf88 00030002 00000005
```

Offset 0 is the cb pointer, so the cb is located at `00650fd8`, and the class name is at offset c. Dump the memory at (`00650fd8 + c`) to show the pointer, then dump the memory at that address in ASCII format; the class name is shown as com/bco/bosc/core/charges/ChargeAdjustment. Now you know the name of the class and the method is known. If these are unique, you do not need to find the method signature. Otherwise, you must find the method signature.

### Finding the method signature

In the example dump that is shown above, the method signature pointer is `12f13518`. Dump this in ASCII format to show the signature:

```
Lcom/bco/bosc/core/BasicOrderData;Lcom/bco/bosc/tools/ChargeType;)Z
```

You now know the exact method that was executing when the JIT code crashed. It is:

```
com.bco.bosc.core.charges.ChargeAdjustment.removeItemChargeAdjustements ()
```

which takes objects BasicOrderData and ChargeType as parameters and returns type boolean. You can start running diagnostics, such as method trace, with this as a start point.

## 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 81 for details.) The required files are:

- JVM-produced Javadump file (Javacore)
- Dr. Watson Log
- Cross-platform dump file (SDFF)

# 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 (LK)" on page 222.

## 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 29, "Using the dump formatter," on page 261 for details.) The jvmdump application is provided as-is. If you would like a copy, e-mail jvmcookbook@uk.ibm.com.

Alternatively, if you have the Microsoft debugging tools installed, you can use Windbg to generate a minidump. To do this:

1. Launch Windbg and attach to the hung JVM process.
2. In the command window, enter:

```
.dump /mfpa /c "dump comment" <dumpname>
```

A minidump of the attached process is produced. You can then translate this minidump into a dump formatter dump in the usual way. (See Chapter 29, "Using the dump formatter," on page 261 for details.)

## Creating a user dump file for a hung process using the Dr. Watson utility

If the JVM appears to be hung and is not responding to signals to get a core dump, use the procedure described below to obtain a Windows user dump file. You can send this dump file to IBM service instead of an IBM core dump, and the service team can extract relevant information.

Before running Java:

1. On the Start menu, click **Run**, type `drwtsn32`, and click **Run**.
2. When the Dr Watson dialog appears, in the **Crash Dump** field type your required destination for the dump; for example, `d:\toibm`. Click **OK**.

After a hang:

1. Find the PID of the Java executable by using the Task Manager (Ctrl-Alt-Del and then click **Task Manager**), click the **Process** tab, and find the PID for java.exe or javaw.exe.
2. From a command line, type:

   ```
   drwtsn32 -p <pid>
   ```

   For example:

   ```
   drwtsn32 - p 868
   ```

   where 868 is the PID you just found.

   This command will generate a file called user.dmp in the location you specified. You can run the **jextract** tool against this dump to get a dump formatter dump, which you can use to do the debugging yourself.

# 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. The Windows NT operating system tends to use addresses `0x77nnnnnn` for its memory segments.

Java uses the following:

```
0x00400000   java executable      (JAVA.EXE)
0x10000000   main java library    (jvm.dll)
0x00b70000   extended HPI library (xhpi.dll)
0x00b80000   HPI library          (hpi.dll)
0x090d0000   java tools library   (java.dll)
0x09100000   java zip library     (zip.dll)
0x091d0000   JIT library          (jitc.dll)
```

Note that the JIT puts compiled code in its segment. The Javadump tells you that the size of the JIT code is, for example, 5000 bytes. Thus you would expect JIT code to occupy memory `0x091d0000` to `0x091d4fff` inclusive. It is not uncommon to see crashes indicating that code failed, for example, at address `0x091d6abc`. This is a sure indication that the crash has happened in JIT-compiled code.

## 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 299 below.)
  - Memory leak internal to JVM.

## Tracing leaks

The dbgmalloc library can be linked in to a customer native library to help identify native memory leaks. dbgmalloc must be linked in to the library before the C-runtime library, so that the standard memory routines can be overridden.

Note that dbgmalloc is meant for IBM use only.

Add this to the DLL command:

```
$SDK\jre\bin\libdbgmalloc.lib
```

The environment variable `$SDK` points to the Java SDK directory (for example, /opt/IBMJava2-142).

Note that the Windows JVM does not support the backtrace trace option, so it will be harder to use the debug malloc library to find a native memory leak.

Other tools for tracing leaks are available. Some of these tools are freeware. It is outside the scope of this book to describe how to use all these tools. Chapter 38, "Using third-party tools," on page 383 describes some of the tools that are available.

Some useful techniques are built into the JVM:

- The **verbose GC** option
- HeapDump: See Chapter 26, "Using Heapdump," on page 245
- HPROF tools

## Verbose GC

Verbose GC is a command-line option that you supply to the JVM at startup time. The format is: **-verbose:gc**

This option switches on a substantial trace of every garbage collection cycle. Output typically looks like this:

```
<GC: Tue Apr 24 10:49:58 2001>
<GC(24): freed 1541416 bytes in 12 ms, 53% free (2248392/4194296)>
<GC(24): mark: 10 ms, sweep: 2 ms, compact: 0 ms>
<GC(24): refs: soft 0 (age >= 32), weak 0, final 116, phantom 0>
```

**Notes:**

1. `GC(24)`: The 24th garbage collection cycle in this JVM.
2. `freed 1541416 bytes`: An indication of the amount of activity since the last garbage collection cycle.
3. `refs: soft 0 ...` — the number of soft, weak, final, and phantom reference objects found in the cycle. Large increasing numbers of references indicate some kind of problem whereby reference objects are pinning down the objects to which they refer.
4. For more information on garbage collection, see Chapter 2, “Understanding the Garbage Collector,” on page 7.

This trace should allow you to see the gross heap usage in every garbage collection cycle. For example, you could monitor the output to see the changes in the free heap space and the total heap space.

## Using HeapDump to debug memory leaks

For details about analyzing the Java Heap, see Chapter 26, “Using Heapdump,” on page 245.

# Debugging performance problems

Performance-related problems occur when:

- Applications consume 100% CPU when not required.
- Unnecessary events that can hinder performance are generated from the virtual machine.
- Memory consumption with JVM is abnormal, but the program seems to be running normally.
- Your application is very slow.

When a Java application seems to be running slowly, you should check the various JIT options and ensure that a suitable JIT compiler exists for the virtual machine before you try anything else. Refer to Chapter 30, “JIT diagnostics,” on page 295.

Use the **hprof** tool, which can help find the CPU usage problems with applications. Different CPU options can be used to identify the method or thread that consumes more CPU time. **Hprof** does not calculate the count of CPU utilization by internal methods, but flattens the hierarchy of the methods and adds the counts to the method that is at a lower level in the stack trace. Refer to **java -Xrunhprof:help** (in Chapter 36, “Using the JVMPI,” on page 369) for further options.

The memory consumption performance issues can be addressed by various garbage collection options. Refer to Chapter 31, “Garbage Collector diagnostics,” on page 299

page 299. Verify that the OS is tuned with sufficient paging memory for Java heap management. The application heap tuning also plays a vital role. Using System.gc() is not a good option because it is totally virtual machine dependent and cannot be used to optimize the memory usage. Instead, your applications should take proper care in managing the memory allocated to different objects. If you do use System.gc(), try making it optionally compilable and switch it off to check if this is impacting your performance. You can find general guidance on good garbage collection practice in Chapter 2, "Understanding the Garbage Collector," on page 7.

Other tools, such as **JProf**, **ProGaurd**, and **JinSight**, can give further inputs on various parameters of a program running in Java. Some of these tools are described in Chapter 38, "Using third-party tools," on page 383.

## Data required for submitting a bug report

IBM service requires:

- Description of performance issue.
- A heapdump (see Chapter 26, "Using Heapdump," on page 245) if you think that you have a memory consumption or thrashing problem.
- Javadump snapshots (see Chapter 25, "Using Javadump," on page 219) of the JVM before performance became a problem and after.
- If performance is a permanent problem, send a couple of snapshots that are separated by approximately 10 minutes, by using the dump extractor (see "General debugging techniques" on page 154) after the point at which performance became a problem.

## Frequently reported problems

IBM service often receives problems that are caused by:

- Garbage collection cycles consuming too much processor time:
  1. System.gc() check. Check for and remove any unwanted System.gc() calls in your code. If you want to use this call, make it conditionally compilable and check whether switching it off addresses performance issues.
  2. Heap management check. If your heap is too small, for example, the Garbage Collector will continually run into allocation faults. Refer to Chapter 31, "Garbage Collector diagnostics," on page 299 and Chapter 2, "Understanding the Garbage Collector," on page 7 for data to help you to set the correct heap size and tune the way garbage collection runs.
- Unused objects are not being collected.

  See "Common causes of perceived leaks" on page 299.
- Heap never shrinks.

  Refer to Chapter 2, "Understanding the Garbage Collector," on page 7 for conditions under which this can occur.

# 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.

- Heapdumps. If generated automatically, they are essential. They are also essential if you have a memory or performance problem.
- Cross-platform dump formatter (SDFF) dump. This is the key to most problems.
- Dr. Watson logs. Send these if the operating system tells you it has generated any.
- WebSphere Application Server logs (see Chapter 13, "Working in a WebSphere Application Server environment," on page 99), if you are working in a WebSphere Application Server environment.
- Other data, as determined by your particular problem.

## Controlling the JVM when used as a browser plug-in

When the JVM operates as a browser plug-in, you cannot *directly* pass parameters into it or collect output. This section describes how to overcome this limitation. The same technique applies to other platforms. Use the appropriate commands to set environment variables and launch the browser.

This example shows you how to set verbosegc and dump options. Start a command session and then:

```
set IBM_JAVA_OPTIONS=-verbose:gc
set
JAVA_DUMP_OPTS="ONERROR(JAVADUMP,SYSDUMP),ONEXCEPTION(JAVADUMP,SYSDUMP),
    ONDUMP(JAVADUMP)"
set IBM_HEAPDUMP=true
set IBM_HEAPDUMP_OUTOFMEMORY=true
set IBM_HEAPDUMPDIR=C:\Dumpdir
set IBM_JAVACOREDIR=C:\Dumpdir
```

Set the **PATH** environment variable to point to your browser if it is not already set. For example:

```
set path=<path to browser executable>;%path%
```

Launch the browser from the command-line session:

```
mozilla>Output.txt 2>&1
```

or

```
start mozilla>Output.txt 2>&1
```

Use the browser as normal. The JVM plug-in will start with the specified environment. The **IBM_JAVA_OPTIONS** environment variable is the key to setting JVM command-line options. It causes the verbosegc output to be written to the Output.txt file in the current working directory. Heapdumps and Javadumps will be dumped to C:\Dumpdir when `OutofMemoryError` occurs.

Alternatively, you can generate JVM command-line options by passing **-verbose:gc** as the JRE runtime parameter in the Plug-in Control Panel and then launching Mozilla as shown above.

# Chapter 19. z/OS problem determination

This chapter describes problem determination on z/OS in:



## Setting up and checking your z/OS environment

### Maintenance

The Java for OS/390 and z/OS website 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 182, LE provides storage from Subpool 2, key 8 for C/C++ programs like the JVM that use C RTL calls like malloc() to obtain memory. The LE HEAP refers to the areas obtained for all C/C++ programs that run in a process address space and request storage.

This area is used for the allocation of the Java heap where instances of Java objects are allocated and managed by Garbage Collection. The area is used also for any underlying allocations that the JVM makes during operations. For example, the JIT compiler obtains work areas for compilation of methods and to store compiled code.

Because the JVM must preallocate the maximum Java heap size so that it is contiguous, the total private area requirement is that of the maximum Java heap size that is set by the -Xmx command line 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 dlls, or failures to complete subcomponent initialization during startup.

## Standalone environment checking utility program

The **jdkiv** utility is a small standalone program that helps solve some of the problems described above. This tool is available on request from jvmcookbook@uk.ibm.com. It prints out the fields of interest from the VSM (Virtual Storage Management) LDA (Local Data Area) in a format similar to the output from the IPCS command:

```
ip verbx vsmdata 'nog,summary,jobname(jjjjjjjj)'
```

This utility is particularly useful for identifying when the private area for a process address space has been limited by an exit or system setting, and is lower than anticipated. The interaction between exits and system settings is quite complex, so it is useful to know what the final values in these LDA fields are.

Sample output from **jdkiv**:

```
jdkiv -- SDK for z/OS install verification program

(a) Operating System Check for installed SDK

CVT address:fc69b8
CVTFLAG2=f8
BFP Hardware Instruction set present
OS is:z/OS V01 R04.00 Machine 9672 Node MVJ1
getrlimit reports RLIMIT_AS as current:-2147483648, max:-2147483648

(b) Virtual Storage check for process 67174503 (0x4010067) ASCB:00f46880, ASID:0098

Virtual Storage Management, Local Data Area for this process at:7ff16ea0
Summary of Key Information from LDA (Local Data Area), (eyecatcher LDA ) :

STRTA  =00006000   (ADDRESS of start of private storage area)
SIZA   =009fa000   (SIZE of private storage area) (10216 K)
CRGTP  =00018000   (ADDRESS of current top of user region)
LIMIT  =009fa000   (Maximum SIZE of user region) (10216 K)
LOAL   =00012000   (TOTAL bytes allocated to user region)
HIAL   =00037000   (TOTAL bytes allocated to LSQA/SWA/229/230 region)
SMFL   =ffffffff   (IEFUSI specification of LIMIT)
SMFR   =ffffffff   (IEFUSI specification of VVRG)

ESTRTA =1bb00000   (ADDRESS of start of extended private storage area)
ESIZA  =64500000   (SIZE of extended private storage area) (1605 MB)
ERGTP  =1c042000   (ADDRESS of current top of extended user region)
ELIM   =7f606000   (Maximum SIZE of extended user region) (2038 MB)
ELOAL  =00542000   (TOTAL bytes allocated to extended user region)
EHIAL  =00c3d000   (TOTAL bytes allocated to extended LSQA/SWA/229/230)
SMFEL  =ffffffff   (IEFUSI specification of ELIM)
SMFER  =ffffffff   (IEFUSI specification of EVVRG)

REGREQ =03600000   (REGREQ) (54 MB)
VVRG   =009fa000   (VVRG)
```

```
EVVRG  =7f606000   (EVVRG)
Total private area above and below capped by REGION limits is 1605 MB
REGION size is adequate for JVM instantiation with default max heap and JIT usage

(c) Envvar Check for installed SDK



No redundant or deprecated environment variables were found set
LIBPATH=/lib:/usr/lib:/usr/lpp/db2610/db2610/lib:.:/u/sovbld/bldsys/ode/os390
Note:LIBPATH is only used by Java for JNI
CLASSPATH=./.:/usr/lpp/db2/db2510/classes/db2jdbcclasses.zip:
PATH=/u/dclarke/cm131s/inst.images/mvs390_oe_1/sdk/jre/bin:
   /u/dclarke/cm131s/inst.images_g/mvs390_oe_1/sdk/jre/bin:
     /usr/lpp/skrb/bin:/bin:/usr/sbin:.:/u/dclarke/J1.3/bin
JAVA_DUMP_OPTS=ONINTERRUPT(ALL) THREAD(ALL) NOFILE NOVARIABLE PAGESIZE(0)
_CEE_RUNOPTS=AL(ON),POS(ON)
Warning:JVM default LE options are as follows
        #pragma runopts(ALL31(ON))
        #pragma runopts(ANYHEAP(2M,512K,ANYWHERE,KEEP))
        #pragma runopts(BELOWHEAP(8K,2K,KEEP))
        #pragma runopts(HEAP(8M,2M,ANYWHERE,KEEP))
        #pragma runopts(LIBSTACK(1K,1K,FREE))
        #pragma runopts(STACK(48K,16K,ANYWHERE,KEEP))
        #pragma runopts(STORAGE(NONE,NONE,NONE,1K))
Any change from these values may cause unexpected results or performance degradation
jdkiv -- there were 1 warning(s) or error condition(s) detected :-(
```

### 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 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. Both of these variables are described in Chapter 27, “JVM dump initiation,” on page 251.

## General debugging techniques

Alongside the debugging tools that are available on all platforms, z/OS also:

- Implements its own Heapdump generation facility.
- Has a currently unsupported debugging toolset that is called svcdump.jar. This toolset contains various packages, for example, the Dump package, as described below, and is available through IBM support.

### Starting Javadumps in z/OS

See Chapter 25, “Using Javadump,” on page 219.

### Starting Heapdumps in z/OS

See Chapter 26, “Using Heapdump,” on page 245.

## The dump tool

You can use the dump tool of svcdump.jar for the analysis of SVC and Transaction Dumps. You can use it instead of IPCS to gain information (for example, tracebacks) from the dump. The Dump tool has the added advantage that it prints Java method names, and understands some underlying JVM structures. The syntax for using the Dump tool is:

```
java -classpath svcdump.jar com.ibm.jvm.svcdump.Dump [-options] <filename>
```

where -options can be one of:

- -cache: Print alloc cache.
- -dis <addr> <n>: Disassemble <n> instructions, starting at <addr> (hex).
- -dump <addr> <n>: Dump<n> words of storage, starting at <addr> (hex).
- -dumpclasses: Dump all the classes and their methods.
- -dumpclass <addr>: Dump the class at <addr>.
- -dumpnative: Dump all the native methods in all the loaded classes.
- -dumpprops: Dump all the system properties.
- -exception: Print old exceptions.
- -heap: Print a summary of objects that are in the heap.
- -systrace: Prints the system trace table.
- -r<n>: Include saved register <n> in the stack trace.
- -verbose: Print extra information.
- -debug: Print debug information.
- -version: Print information about the version.

The default output, if no options are specified, consists of, for every valid address space, a listing of all the TCBs that are in that address space. The traceback and the trace table, if available, is listed for each TCB. If available, the trace table is obtained from the System Trace entries for that TCB, and the addresses that are found are converted into a function name by finding the closest function entry point. The trace lists the function names that have been found. The names are listed in the sequence of which functions feature most often. Many entries of unknown function (address) might be present. This means that the function name for that address was unknown, possibly because it is Kernel code. This gives an indication of what the TCB has been running previously. The environment variables and the dll table for this address space are also printed out. The expected tracebacks should be of the form of:

```
found trace table in asid 0
found trace table in asid 1
found trace table in asid 6
found trace table in asid c9e7d3c3
found trace table in asid c9e7c3c4
found trace table in asid c9e7c3c1
found trace table in asid c9c1d9c3
found trace table in asid e2e8e2e9
found trace table in asid 4
found Linkage TCB af1bc0 tid 0c6c3880 caa 00016a88
Dsa Entry Offset Function
---------------------
0c330848 07193098 058b3d00 select1
0c330750 0d0ff6c0 000002da ThreadUtils_BlockingSection
0c330568 0d0ea088 000001da sysTimeout
0c3304a8 0d0da110 00000088 sysRead
0c3303f0 0ce05e20 00000120 JVM_Read
0c330340 0024af58 0000018e readBytes
0c330288 0022b618 00000096 Java_java_io_FileInputStream_readBytes
```

```
0c330198 0ca179e8 00000102
java/io/FileInputStream.readBytes(Ljava/io/FileDescriptor;)I
0c3300a0 0d01d008 00000536 mmipSelectInvokeJavaMethod
(java/io/FileInputStream.read)
0c32ffa0 0d01d008 00000536 mmipSelectInvokeJavaMethod
(java/io/BufferedInputStream.read1)
0c32fea0 0d01d0f0 0000044e mmipSelectInvokeSynchronizedJavaMethod
(java/io/BufferedInputStream.read)
0c32fdb8 0d01d008 00000536 mmipSelectInvokeJavaMethod
(java/io/FilterInputStream.read)
0c32fcc0 0d01d008 00000536 mmipSelectInvokeJavaMethod
(java/io/InputStreamReader.fill)
0c32fbc0 0d01d008 00000536 mmipSelectInvokeJavaMethod
(java/io/InputStreamReader.read)
0c32fad0 0d01d008 00000536 mmipSelectInvokeJavaMethod
(java/io/BufferedReader.fill)
0c32f9b0 0ca1c994 00000118 java/io/BufferedReader.readLine(Z)Ljava/lang/String;
0c32f8d0 0d01d008 00000536 mmipSelectInvokeJavaMethod
(java/io/BufferedReader.readLine)
0c32f7a0 0d01d008 00000536 mmipSelectInvokeJavaMethod
(com/ibm/ctg/server/JGate.main)
0c32f6d0 0d01ca88 00000ab6 INVOKDMY
0c32f600 0d01bf48 00000098 EXECJAVA (^X ^OXá.^L2(F)
0c32f550 0d03c2f0 000000ee mmipExecuteJava (com/ibm/ctg/server/JGate.main)
0c32f428 0d041158 000006ce xeRunJniMethod
0c32f338 0cdd37c0 000001a8 jni_CallStaticVoidMethod
0c32f1e0 0c3088b8 00001c30 main
0c32f0c8 072c4bae f8d5297a EDCZMINV (main invocation event handler)
0c32f018 0000e2c0 0000013e CEEBBEXT (bootstrap routine)
00017330 0c308738 0000b19e (unknown)
00005f88 00000000 00fd2500 (unknown)
found LE TCB ad7b60 tid 0c88d798 caa 0c34b408
Dsa Entry Offset Function
---------------------
0c34ca38 073b4498 f8c4bb68 CEEOPCW
0c34c978 071449d8 0000007a pthread_cond_wait
0c34c880 0d0ff6c0 000001a2 ThreadUtils_BlockingSection
0c34c788 0d0e1ac8 000001ac sysSignalWait
0c34c6b8 0d05e730 000000b6 signalDispatcherThread
0c34c608 0d060d08 000001d0 xmExecuteThread
0c34c560 0d04dc30 0000006e threadStart
0c34c3f8 0d0fffc8 00000a3a ThreadUtils_Shell
```

## The -cache option

This option prints the Java alloc cache as:

```
alloc cache info:
cache_busy = 0x0
cache_size = 0x7bbc
cache_block = 0x14208ef0
cache_orig_size = 0x10004
14210aac: len = 20 methods = efd7ee0 flags = 0 class = java/lang/String
14210acc: len = 70 methods = 32 flags = 2a
14210b3c: len = 68 methods = 2c flags = 2a
14210ba4: len = 20 methods = efd7ee0 flags = 0 class = java/lang/String
14210bc4: len = 20 methods = 110c4d00 flags = 0 class = java/lang/ref/Finalizer
14210be4: len = 20 methods = 110c7560 flags = 0 class = java/lang/ClassLoader$NativeLibrary
14210c04: len = 20 methods = efd7ee0 flags = 0 class = java/lang/String
14210c24: len = 50 methods = 1f flags = 2a
14210c74: len = 20 methods = efd7ee0 flags = 0 class = java/lang/String
```

This gives an idea of the most recently allocated objects, because they are still in the alloc cache.

## The -exception option

This option forces the output of any leftover exception objects. (Pending exceptions are printed by default.) It prints the last exception that was thrown by each thread (if any) in terms of the exception class plus any additional detail:

```
found old exception: java/lang/NoSuchMethodError: setInternalError
```

## The -dis <addr> <n> option

This option disassembles the instruction at the hex address <addr> and the next <n> instructions. The disassembler is not complete, but does know about the most common instructions. When it finds an instruction that it does not understand, it throws an exception and exits. This usually occurs when the disassembler reaches the end of a function.

```
Disassembly starting at 0x11a90c48
0x11a90c48: (0x00000000): B      x'22'($r15)
0x11a90c6a: (0x00000022): STM    $r14,$r11,x'c'($r13)
0x11a90c6e: (0x00000026): L      $r14, x'4c'($r13)
0x11a90c72: (0x0000002a): LA     $r0, x'450'($r14)
0x11a90c76: (0x0000002e): CL     $r0, x'314'($r12)
0x11a90c7a: (0x00000032): LA     $r3, x'3a'($r15)
0x11a90c7e: (0x00000036): BGT    x'14'($r15)
0x11a90c82: (0x0000003a): L      $r15, x'280'($r12)
0x11a90c86: (0x0000003e): STM    $r15,$r0,x'48'($r14)
0x11a90c8a: (0x00000042): MVI    x'0'($r14), x'10'
0x11a90c8e: (0x00000046): ST     $r13, x'4'($r14)
0x11a90c92: (0x0000004a): LR     $r13, $r14
0x11a90c94: (0x0000004c): L      $r4, x'1f4'($r12)
0x11a90c98: (0x00000050): L      $r5, x'7ae'($r3)
0x11a90c9c: (0x00000054): LA     $r2, x'c4'($r13)
0x11a90ca0: (0x00000058): LA     $r1, x'98'($r13)
0x11a90ca4: (0x0000005c): ST     $r2, x'98'($r13)
0x11a90ca8: (0x00000060): L      $r14, x'170'($r5,$r4)
0x11a90cac: (0x00000064): LM     $r15,$r0,x'8'($r14)
0x11a90cb0: (0x00000068): ST     $r0, x'1f4'($r12)
0x11a90cb4: (0x0000006c): BALR   $r14, $r15
0x11a90cb6: (0x0000006e): L      $r7, x'7b2'($r3)
0x11a90cba: (0x00000072): L      $r6, x'174'($r5,$r4)
0x11a90cbe: (0x00000076): LA     $r0, x'f4'($r13)
0x11a90cc2: (0x0000007a): ST     $r0, x'ac'($r13)
0x11a90cc6: (0x0000007e): LA     $r1, x'407'($r7)
0x11a90cca: (0x00000082): LA     $r14, x'fc'($r13)
0x11a90cce: (0x00000086): LA     $r10, x'234'($r13)
0x11a90cd2: (0x0000008a): ST     $r1, x'9c'($r13)
0x11a90cd6: (0x0000008e): LA     $r11, x'41c'($r7)
0x11a90cda: (0x00000092): LM     $r15,$r0,x'8'($r6)
0x11a90cde: (0x00000096): ST     $r10, x'98'($r13)
0x11a90ce2: (0x0000009a): ST     $r11, x'438'($r13)
0x11a90ce6: (0x0000009e): ST     $r11, x'a0'($r13)
0x11a90cea: (0x000000a2): ST     $r2, x'a4'($r13)
0x11a90cee: (0x000000a6): LA     $r1, x'98'($r13)
0x11a90cf2: (0x000000aa): ST     $r14, x'a8'($r13)
0x11a90cf6: (0x000000ae): ST     $r0, x'1f4'($r12)
0x11a90cfa: (0x000000b2): BALR   $r14, $r15
```

## The -dump <addr> <n> option

This option dumps <n> words of storage starting at the hex address <addr>. Currently it is not possible to specify which of the address spaces in the dump to use. The tool defaults to the first Java address space.

## The -r<n> option

This option includes the saved register <n> that is in the stack trace. It assumes that the register value is held in the DSA.

```
found Usta TCB a7e288 tid 1ac30e20 caa 109f5120
Dsa         Entry       Offset      r12         Function
---         -----       ------      ---         --------
10a06d90    11aa58d8    fff8909a    109f5120    SYSTDUMP
10a06940    11a90c48    00000414    109f5120    ThreadUtils_CoreDump
10a06830    11a734b0    000004b2    0ef473a0    userSignalHandler
10a06780    11a73a18    000000b8    40404040    intrDispatch
10a066c8    061596b8    000000c4    40404040    @@GETFN
10a06068    0628af48    0000075e    109f5120    __zerros
10a034d0    00000008    0638260e    109f5120    null
10a02a70    11a566d0    fefac3f8    00000000    CompareAndSwap_Impl
10a029c0    119223f0    000000a2    109f5120    pin_object
10a02918    11740c40    00000114    109f5120    jni_GetPrimitiveArrayElements
10a02710    1af1a388    000000b2    109f5120    MVS_CcicsInit
10a02660    1af143b8    000000b0    109f5120    Java_com_ibm_ctg_server_ServerECIRequest_CcicsInit
10a025a8    06419078    0000005c    109f5120    CEEPGTFN
10a02118    119a20d0    00000138    109f5120    MMIPSJNI
10a02038    1199fd48    000003ce    109f5120    mmisInvokeJniMethodHelper
(com/ibm/ctg/server/ServerECIRequest.CcicsInit)
10a01f68    1198e3f8    00000100    109f5120    mmipInvokeJniMethod
(com/ibm/ctg/server/ServerECIRequest.CcicsInit)
```

## Using IPCS commands

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 ledata 'cee,asid(007d),tcb(tttttt)'**
This command formats out the traceback for one specific thread.

**ip summ regs asid(x'007d')**
This command formats out the TCB/RB structure for the address space. It is rarely useful for JVM debugging.

**ip verbx sumdump**
Then issue find 'slip regs sa' to locate the GPRs and PSW at the time a SLIP TRAP s 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.

### 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 request an unformatted transaction dump through the BCP IEATDUMP service and to produce a formatted dump of internal JVM state, which is known as the Javadump. Output should be written to the message stream that is written to stderr in the form of:

```
JVMHP001:JVM signal handler receives signal number 3 (SIGABRT)
JVMHP002:JVM requesting Transaction Dump
JVMHP007:JVM default dump data/set name pattern for IEATDUMP was RICCOLE.SYSTDUMP...
JVMHP005:Complete Transaction dump was written in 20566ms
JVMDG217: Dump Handler is Processing a Signal - Please Wait.
JVMHP002: JVM requesting System Transaction Dump
JVMHP012: System Transaction Dump written to RICCOLE.SYSTDUMP.D030929.T134331
JVMDG303: JVM Requesting Java core file
JVMDG304: Java core file written to /u/riccole/JAVADUMP.20030929.134401.16908553
.txt
JVMDG215: Dump Handler has Processed Error Signal 3.
CEE5207E The signal SIGABRT was received.
JVMXM004:JVM is performing abort shutdown sequence
JVMDG303:JVM writing JAVADUMP file
JVMDG304:JAVADUMP written to /u/riccole/JAVADUMP.20030929.134401.16908553
.txt
```

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 24, "Overview of the available diagnostics," on page 213 and Chapter 27, "JVM dump initiation," on page 251.

These two documents (that is, the Javadump file and the transaction dump) 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

Any one of the three documents that you gathered, (see "Documents to gather" on page 174) should be enough to determine the failing function, and therefore determine to which IBM support group the problem should be routed, or whether application native C code is at fault.

The most practical way to find where the exception occurred is to review either the CEEDUMP 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 IPCS LEDATA VERB Exit, or the svcdump.jar toolset. These generate a report that is similar to the CEEDUMP.

In each case, the report shows the C-Stack (or native stack, which is separate from the Java stack that is built by the JVM because one method gives control to another). 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 CEEDUMP shows where a failure occurred:

```
Traceback:
    DSA Addr  Program Unit  PU Addr   PU Offset  Entry         E Addr    E  Offset   Statement  Load Mod  Service  Status
    196784C0  /u/sovbld/hm131s/hm131s-20020716/src/hpi/pfm/threads_utils.c
                            1A2FF8A0  +000006AA  ThreadUtils_CoreDump
                                                                1A2FF8A0  +000006AA        1662  *PATHNAM  h020716  Call
    196783B0  /u/sovbld/hm131s/hm131s-20020716/src/hpi/pfm/interrupt_md.c
                            1A2E2108  +000004B0  userSignalHandler
                                                                1A2E2108  +000004B0         376  *PATHNAM  h020716  Call
    19678300  /u/sovbld/hm131s/hm131s-20020716/src/hpi/pfm/interrupt_md.c
                            1A2E2670  +000000B6  intrDispatch  1A2E2670  +000000B6         642  *PATHNAM  h020716  Call
    19678248                084F8B98  +0000001A  @@GETFN       084F8AF0  +000000C2              CEEEV003            Call
    19677CD8                08441918  +0000073A  __zerros      08441918  +0000073A              CEEEV003  UQ42798  Call
    19675968  CEEHDSP        0868E4E8  +00002B84  CEEHDSP       0868E4E8  +00002B84              CEEPLPKA  UQ47631  Call
    196753E0                1A21D3C8  +000004E0  mmipSelectInvokeJavaMethod
                                                                1A21D3C8  +000004E0              *PATHNAM            Exception
    19675300                1A21D3C8  +00000534  mmipSelectInvokeJavaMethod
                                                                1A21D3C8  +00000534              *PATHNAM            Call
    19675208                1A21D3C8  +00000534  mmipSelectInvokeJavaMethod
                                                                1A21D3C8  +00000534              *PATHNAM            Call
    19675110                1A21D3C8  +00000534  mmipSelectInvokeJavaMethod
                                                                1A21D3C8  +00000534              *PATHNAM            Call
    19675020                1A21D3C8  +00000534  mmipSelectInvokeJavaMethod
                                                                1A21D3C8  +00000534              *PATHNAM            Call
    19674F48                1A21D3C8  +00000534  mmipSelectInvokeJavaMethod
                                                                1A21D3C8  +00000534              *PATHNAM            Call
    19674E78                1A21CE48  +00000AB4  INVOKDMY      1A21CE48  +00000AB4              *PATHNAM            Call
    19674DA8                1A21C308  +00000096  EXECJAVA      1A21C308  +00000096              *PATHNAM            Call
    19674CF8  /u/sovbld/hm131s/hm131s-20020716/src/jvm/pfm/xe/mmi/mmi_supp
                            1A23C6B0  +000000EC  mmipExecuteJava
                                                                1A23C6B0  +000000EC         149  *PATHNAM  h020716  Call
    19674BD0  /u/sovbld/hm131s/hm131s-20020716/src/jvm/sov/xe/common/run.c
                            1A242940  +00000494  xeRunDynamicMethod
                                                                1A242940  +00000494         892  *PATHNAM  h020716  Call
    19674B20  /u/sovbld/hm131s/hm131s-20020716/src/jvm/sov/ci/jvm.c
                            1A003ED0  +000000E0  threadRT0     1A003ED0  +000000E0         332  *PATHNAM  h020716  Call
    19674A70  /u/sovbld/hm131s/hm131s-20020716/src/jvm/sov/xm/thr.c
                            1A2610C8  +000001CE  xmExecuteThread
                                                                1A2610C8  +000001CE        1433  *PATHNAM  h020716  Call
    196749C8  /u/sovbld/hm131s/hm131s-20020716/src/jvm/pfm/xe/common/xe_th
                            1A24DFF0  +0000006C  threadStart   1A24DFF0  +0000006C          79  *PATHNAM  h020716  Call
    19674860  /u/sovbld/hm131s/hm131s-20020716/src/hpi/pfm/threads_utils.c
                            1A300FC8  +00000A38  ThreadUtils_Shell
                                                                1A300FC8  +00000A38         900  *PATHNAM  h020716  Call
    196747A8                084F8B98  +0000001A  @@GETFN       084F8AF0  +000000C2              CEEEV003            Call
    7F6B49C0  /u/sovbld/hm131s/hm131s-20020716/src/tools/sov/java.c
                            19508808  -194FCE4A  main          19508808  -194FCE4A              *PATHNAM  h020716  Call
```

**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 `mmipSelectInvokeJavaMethod`.

2. The value under Service for each DSA is the service string. The string is built in the format of `.cyymmdd`, where `c` 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. Similarly, any program unit whose z/OS UNIX filename begins with `/u/sovbld` 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.
3. The value for Entry gives the name of the function that is being executed in that DSA. The EXECJAVA function, or functions that start with `mmi` are part of the JVMs Mixed Mode Interpreter, which written in 390 Assembler for z/OS. Multiple entries into these functions are normal. The same functions are entered repeatedly as the Interpreter interprets bytecode in different methods. Functions that appear as a complete package name or `class name.method name(method signature)` in slash form are JIT-compiled methods. The JIT Compiler has taken the bytecode for this method and compiled it into a native binary. When the native binary is emitted, it is given LE PPA areas that identify the function name as being the fully-qualified package name or `class name.method name(method signature)`. For example:

   ```
   java/io/BufferedReader.readLine(Z)Ljava/lang/String;
   ```

   Functions that are in the JVM itself are often identified by a two character prefix. For example:

   - dc=data conversion
   - xm=execution management
   - xe=execution environment
   - cl=class loader
   - st=storage management (including object allocation and garbage collection)
   - lk=locking

   Functions that are named according to the Java format are native methods. For example, function `Java_java_io_FileInputStream_readBytes` is the C function name that is created to support native method `java/io/FileInputStream.readBytes`. Functions that start with `G3*` or `jitc*` are part of the JIT compiler.

If the Dump tool is used from the svcdump.jar package, the name of the Java method that is being executed by the MMI in each stack frame is displayed. The Dump tool also displays the values of the parameters that are being passed to each function if they are available from the dump.

## 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.

The normal way to inspect a TDUMP is by using IPCS (see "Using IPCS commands" on page 173). You can also inspect a TDUMP using a Java application such as svcdump, or jformat, if the dump data set has been transferred in binary mode to the inspecting system.

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. If the record ids are of the type DR2 in columns 1 through 3, you need an OS/390 V2R10 or z/OS 1.0 or later system IPCS level.
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 (untersed) 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 24AC2408
  09-0094 05C04E50   SRB       070C0000 8AA709B8  00000094 02CC90C0 02CC90EC
                                                  009DFE88 A0
  09-0094 05C04E50   PC    ...   0      0AA70A06           0030B
  09-0094 00000000   SSRV  132          00000000  0000E602 00002000 7EF16000
                                                  00940000
```

For suspected loops you might need to concentrate on ASID and exclude any branch tracing:

```
ip systrace asid(x'3c') exclude(br)
```

**ip summ format asid(x'94')**
To find the list of TCBs, issue a find command for "T C B".

**ip verbx ledata 'ceedump asid(94) tcb(009DFE88)**
Obtains a traceback for the specified TCB.

**ip omvsdata process detail asid(x'94')**
Shows a USS perspective for each thread.

**ip verbx vsmdata 'summary noglobal'**
Provides a summary of the local data area:

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

   L O C A L   S T O R A G E   D A T A   S U M M A R Y


 LOCAL STORAGE MAP
 ____________________________,
|                            |80000000  <- TOP OF EXT. PRIVATE
| Extended                   |
| LSQA/SWA/229/230           |
|____________________________|7F699000  <- ELSQA BOTTOM
|                            |
| (Free Extended Storage)    |11A00000  <- MAX EXT. USER REGION ADDRESS
|____________________________| FD1E000  <- EXT. USER REGION TOP
|                            |
| Extended User Region       |
|____________________________| FA00000  <- EXT. USER REGION START
:                            :
: Extended Global Storage    :
=======================================<- 16M LINE
: Global Storage             :
:____________________________:  A00000  <- TOP OF PRIVATE
|                            |
| LSQA/SWA/229/230           |
|____________________________|  997000  <- LSQA BOTTOM
|                            |
| (Free Storage)             |  60F000  <- MAX USER REGION ADDRESS
```

```
  |_____________________________|   6D000  <- USER REGION TOP
  |                             |
  | User Region                 |
  |_____________________________|    5000  <- USER REGION START
  : System Storage              :
  :_____________________________:       0


Region,Requested         =>          5FA000
IEFUSI/SMF Specification => SMFL : FFFFFFFF  SMFEL: FFFFFFFF
                            SMFR : FFFFFFFF  SMFER: FFFFFFFF
Actual Limit             => LIMIT:   60A000  ELIM :  2000000



STRTA  =     5000  (ADDRESS of start of private storage area)
SIZA   =   9FB000  (SIZE of private storage area)
CRGTP  =    6D000  (ADDRESS of current top of user region)
LIMIT  =   60A000  (Maximum SIZE of user region)
LOAL   =    68000  (TOTAL bytes allocated to user region)
HIAL   =    57000  (TOTAL bytes allocated to LSQA/SWA/229/230 region)
SMFL   = FFFFFFFF  (IEFUSI specification of LIMIT)
SMFR   = FFFFFFFF  (IEFUSI specification of VVRG)

ESTRA  =  FA00000  (ADDRESS of start of extended private storage area)
ESIZA  = 70600000  (SIZE of extended private storage area)
ERGTP  =  FD1E000  (ADDRESS of current top of extended user region)
ELIM   =  2000000  (Maximum SIZE of extended user region)
ELOAL  =   312000  (TOTAL bytes allocated to extended user region)
EHIAL  =   952000  (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:
PSW=078D1400,B258B108
    (Running in PRIMARY  key 8  AMODE 31  DAT ON)
    DISABLED,FOR PER
   ASID(X'0048') 3258B108. SPECIALNAME+24D050 IN EXTENDED PRIVATE
  +0000, 61A4A299  61939797  619181A5  8161C9C2  | /usr/lpp/java/IB |
  +0010, D461D1F1  4BF36182  89956183  9381A2A2  | M/J1.3/bin/class |
  +0020, 89836193  898291A5  944BA296            | ic/libjvm.so     |
   ASID(X'0048') 3258B108. AREA(Subpool252Key00)+24D108 IN EXTENDED PRIVATE
  ASCB72 at F5E100  JOB(TPCCBA0C)  for the home ASID
  ASXB72 at 9FDE90 and TCB72E at 9EC1B0 for the home ASID
  HOME,ASID: 0048 PRIMARY ASID: 0048 SECONDARY ASID: 0048

  General purpose register values
      0-3  00000000  040C0000  0007DA28  3258B06A
      4-7  3258E778  32691028  00012548  0007E198
     8-11 32943C00  3269A088  0000000D  0007E1A5
    12-15 00026A00  0007EB30  3269C5C0  0000000D

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  0EB175B2  00000000

  Control register values
      0-3  5FB5FE50  6B09B07F  6B020640  00C00048
```

```
 4-7  00000048  6FE1E200  FE000000  6B09B07F
 8-11 00000000  00000000  00000000  00000000
12-15 6F9F4E6B  6B09B07F  FF8EFE20  7F938010
```

**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(53) tcb(0088B288)'**
Finds the PSW and registers at time of the exception:

```
MCH_EYE:ZMCH
MCH_GPR00:00000000      MCH_GPR01:00000000
MCH_GPR02:2ABD59B0      MCH_GPR03:28F5C76A
MCH_GPR04:136967C0      MCH_GPR05:0000F588
MCH_GPR06:00000FF8      MCH_GPR07:136A32E0
MCH_GPR08:00000000      MCH_GPR09:2ABD519C
MCH_GPR10:2ABD59B0      MCH_GPR11:11FE8AC8
MCH_GPR12:13E956F0      MCH_GPR13:13EA18C0
MCH_GPR14:808A8887      MCH_GPR15:00000000
MCH_PSW:078D0400 A8F5D1C4     MCH_ILC:0000      MCH_IC1:00
```

**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 ttyp0003  2:34 java -classpath .Test2Frame
```

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

For an explanation of deadlocks and how the Javadump tool is used to diagnose them, see "Locks, monitors, and deadlocks (LK)" on page 222.

### The process is looping

If no deadlock exists between threads and the process appears to be hanging but is consuming CPU time, look at what 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 184 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.

The dbgmalloc library can be linked in to a customer native library to help identify native memory leaks. dbgmalloc must be linked in to the library before the C-runtime library, so that the standard memory routines can be overridden.

Note that dbgmalloc is meant for IBM use only.

Add this option to the c++ command

```
$SDK/J1.4/bin/libdbgmalloc.so
```

The environment variable `$SDK` points to the Java SDK directory (for example, /opt/IBMJava2-142).

### 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 z/OS has 31-bit addressing that allows the addressing of 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.

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 parameter 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 that corresponds to the error codes that are given in Appendix F, "Messages and codes," on page 415.

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 heapdump, and switch on verbosegc output.

The -verbosegc (-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. The verbosegc output also supplies an action value. The number that is associated with action determines the type of garbage collection that is being done:

action=1 means a preemptive garbage collection cycle.

action=2 means a full allocation failure.

action=3 means that a heap expansion takes place.

action=4 means that all known soft references are cleared.

action=5 means that stealing from the transient heap is done.

action=6 means that free space is very low.

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 in required to free memory when it is no longer required. A Java developer is required to removed 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 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

Check whether the JIT compiler is activated. To do this, ensure that:

- You have not unset the environment variable **JAVA_COMPILER**
- You have set the environment variable **JAVA_COMPILER** to something other than *jitc*.
- You have set the system property **-Djava.compiler** to null.

The JIT compiler makes a significant difference to performance. Do not disable it unless under the direction of IBM Service. All areas of JIT optimization are individually switchable, and the JIT allows for selective disablement of compilation for identified methods, so you should always be able to bypass a problem without disabling the JIT compiler completely.

Check whether the system is tuned to cope with the Java managed heap size that you have specified. If the Java managed heap size is large, on a system without large amounts of real storage you might see a performance degradation caused by excessive paging.

If the system intermittently sees high CPU usage for the process in which Java is running, this might be a symptom of excessive garbage collection pauses. The garbage collector is a ″Stop The World″ type, and collection cycles are normally so short (from 5-500 milliseconds, for example) that they are not observed externally. If the collection cycle takes longer for some reason, or occurs more frequently than expected, this will be observed as high CPU. This is because the garbage collection code is CPU-intensive, and the collector uses helper threads for marking objects. These helper threads could possibly be running on all available CPUs. In addition, some mutator threads might be in short ″busy waits″ for the cycle to end. In this case, turn on switch **-verbose:gc** to see how often the cycles are occurring and what their duration is. Pause times over several seconds are worth further investigation. You should also use switch **-Xgcpolicy:optavgpause** to activate JVM use of concurrent marking, to reduce and smooth out pause times, at some small reduction in overall throughput. If this does not help resolve the problem, contact IBM Service before gathering more information.

In benchmark tests, the performance of later releases of the SDK is in general improved over the 1.1.8 SDK. If you experience performance degradation in moving between 1.1.8 and a later release such as 1.4.1, try to narrow this down to a particular Java application before contacting IBM Service.

## 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 IPCS (Interactive Problem Control System).
- CEEDUMP - formatted application level dump, requested by the LE service CEE3DMP.
- JAVADUMP - formatted internal state data produced by the IBM JVM.
- Binary or formatted trace data from the JVM internal high performance trace.
- Debugging messages written to stderr (for example, the output from the JVM when switches like **-verbose:gc** or **-verbose** 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).

The JVM on z/OS makes use of the IEATDUMP service to capture unformatted dumps. These dumps can then be processed with IPCS on OS/390 or z/OS. The internal high performance trace allows for the creation of binary trace files, which can be post-processed on any platform that supports Java.

# Chapter 20. Debugging the ORB

One of the first tasks that you must do 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

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, for example, setting the environment variable JAVA_COMPILER=none. Alternatively, when you are running the application in debugging mode (not in production mode), include the property as **java -Djava.compiler=NONE myapp**.

## 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 7. Packaging*

| | **IBM Platforms** | **Non-IBM Platform** |
|---|---|---|
| Runtime classes | jre/lib/ibmorb.jar | jre/lib/endorsed/ibmorb.jar |
| Tools classes | lib/tools.jar | lib/ibmtools.jar |
| CORBA API classes | jre/lib/ibmorbapijar | jre/lib/endorsed/ibmorbapijar |
| Runtime support | None | jre/lib/endorsed/ibmext.jar |
| rmic wrapper | None | ibm_bin/rmic<br>ibm_bin/rmic.bat |
| idlj wrapper | None | ibm_bin/idlj<br>ibm_bin/idlj.bat |

## 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 on page 188 and 3 on page 188 are alternative methods for reaching the same class.

## 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 turned on; if you set it to **message**, only messages are turned on. Any other value, or no value, turns on traces and messages. The only way not to set this property is not to specify it. A value of false enables it anyway. When enabling any kind of tracing, it is safe to turn this property on.
- **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.DDMMYYYY.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 D, "CORBA minor codes," on page 405 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.

## Java2 security permissions for the ORB

When running with a Java 2 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 8. 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 8. 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.iiop.CDRInputStream.read_value(CDRInputStream.java:1429)
         at com.ibm.rmi.io.ValueHandlerImpl.read_Array(ValueHandlerImpl.java:625)
         at com.ibm.rmi.io.ValueHandlerImpl.readValueInternal(ValueHandlerImpl.java:273)
         at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:189)
         at com.ibm.rmi.iiop.CDRInputStream.read_value(CDRInputStream.java:1429)
         at com.ibm.ejs.sm.beans._EJSRemoteStatelessPmiService_Tie._invoke(_EJSRemoteStatelessPmiService_Tie.java:613)
         at com.ibm.CORBA.iiop.ExtendedServerDelegate.dispatch(ExtendedServerDelegate.java:515)
         at com.ibm.CORBA.iiop.ORB.process(ORB.java:2377)
         at com.ibm.CORBA.iiop.OrbWorker.run(OrbWorker.java:186)
         at com.ibm.ejs.oa.pool.ThreadPool$PooledWorker.run(ThreadPool.java:104)
         at com.ibm.ws.util.CachedThread.run(ThreadPool.java:137)
```

### 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:

```
16:02:33.978 com.ibm.rmi.util.Version logVersions:88 P=953197:O=0:CT ORBRas[default] IBM Java ORB build cndev-20030114
```

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:

```
14:54:14.848 com.ibm.rmi.iiop.Connection <init>:504 LT=0:P=650241:O=0:port=1360 ORBRas[default] Entry
.....
14:54:14.857 com.ibm.rmi.iiop.Connection <init>:539 LT=0:P=650241:O=0:port=1360 ORBRas[default] Exit
```

In this case, the constructor (that is, <init>) of the class Connection is 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:O=0:CT
Local Port:     4899 (0x1323)
Local IP:       9.20.178.136
Remote Port:    4893 (0x131D)
Remote IP:      9.20.178.136
GIOP Version:   1.2
Byte order:     big endian

Fragment to follow: No     // This is the last fragment of the request
Message size:   276 (0x114)
--

Request ID:         5  // Request Ids are in ascending sequence
Response Flag:      WITH_TARGET // it means we are expecting a reply to this request
Target Address:     0
Object Key:         length = 26 (0x1A)  // the object key is created by the server when exporting
                                  // the servant and retrieved in the IOR using a naming service
                    4C4D4249 00000010 14F94CA4 00100000
                    00080000 00000000 0000
Operation:          message // That is the name of the method that the client invokes on the servant
Service Context:    length = 3 (0x3)   // There are three service contexts

Context ID:         1229081874 (0x49424D12) // Partner version service context. IBM only
Context data:       length = 8 (0x8)
                    00000000 14000005


Context ID:         1 (0x1)  // Codeset CORBA service context
Context data:       length = 12 (0xC)
                    00000000 00010001 00010100

Context ID:         6 (0x6)  // Codebase CORBA service context
Context data:       length = 168 (0xA8)
                    00000000 00000028 49444C3A 6F6D672E
                    6F72672F 53656E64 696E6743 6F6E7465
                    78742F43 6F646542 6173653A 312E3000
                    00000001 00000000 0000006C 00010200
                    0000000D 392E3230 2E313738 2E313336
                    00001324 0000001A 4C4D4249 00000010
                    15074A96 00100000 00080000 00000000
                    00000000 00000002 00000001 00000018
                    00000000 00010001 00000001 00010020
                    00010100 00000000 49424D0A 00000008
                    00000000 14000005
Data Offset:        11c
// raw data that goes in the wire in numbered rows of 16 bytes and the corresponding ASCII
decoding
0000: 47494F50 01020000 00000114 00000005    GIOP............
0010: 03000000 00000000 0000001A 4C4D4249    ............LMBI
0020: 00000010 14F94CA4 00100000 00080000    ......L.........
```

```
0030: 00000000 00000000 00000008 6D657373    ............mess
0040: 61676500 00000003 49424D12 00000008    age.....IBM.....
0050: 00000000 14000005 00000001 0000000C    ................
0060: 00000000 00010001 00010100 00000006    ................
0070: 000000A8 00000000 00000028 49444C3A    ...........(IDL:
0080: 6F6D672E 6F72672F 53656E64 696E6743    omg.org/SendingC
0090: 6F6E7465 78742F43 6F646542 6173653A    ontext/CodeBase:
00A0: 312E3000 00000001 00000000 0000006C    1.0............l
00B0: 00010200 0000000D 392E3230 2E313738    ........9.20.178
00C0: 2E313336 00001324 0000001A 4C4D4249    .136...$....LMBI
00D0: 00000010 15074A96 00100000 00080000    ......J.........
00E0: 00000000 00000000 00000002 00000001    ................
00F0: 00000018 00000000 00010001 00000001    ................
0100: 00010020 00010100 00000000 49424D0A    ... ........IBM.
0110: 00000008 00000000 14000005 00000000    ................
```

**Note:** The italic comments that start with a double slash have been added for clarity; they are not part of the traces.

In this example trace, you can see a summary of the principal fields that are contained in the message, followed by the message itself as it goes in the wire. In the summary are several field name-value pairs. Each number is in hexadecimal notation.

Appendix C, "CORBA GIOP message format," on page 401 gives details of the structure of a GIOP message. See also 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-iiop session.

The trace shows that the message is an outgoing request. Therefore, this trace is a client trace, or at least part of the trace where the application acts as a client.

Time information and host names are reported in the header of the message.

The Request ID and the Operation ("message" in this case) fields can be very helpful when multiple threads and clients destroy the logical sequence of the traces.

The GIOP version field can be checked if different ORBs are deployed. If two different ORBs support different versions of GIOP, the ORB that is using the more recent version of GIOP should fall back to a common level. By checking that field, however, you can easily check whether the two ORBs speak the same language.

## Service contexts

The header also records three service contexts, each consisting of a context ID and context data. A service context is extra information that is attached to the message

for purposes that can be vendor-specific (such as the IBM Partner version that is described in the IOR in Chapter 5, "Understanding the ORB," on page 41).

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.

### Hanging

One of the worst conditions is the hanging of client, or server, or both. 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 220.

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)
```

```
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 (Windows NT: tnameserv.exe) 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.TCPTransport.getConnection(TCPTransport.java:178)
             at com.ibm.rmi.iiop.TransportManager.get(TransportManager.java:79)
             at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:131)
             at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:98)
             at com.ibm.CORBA.iiop.ClientDelegate._createRequest(ClientDelegate.java:2096)
             at com.ibm.CORBA.iiop.ClientDelegate.createRequest(ClientDelegate.java:1264)
             at com.ibm.CORBA.iiop.ClientDelegate.createRequest(ClientDelegate.java:1177)
             at com.ibm.rmi.corba.InitialNamingClient.resolve(InitialNamingClient.java:252)
             at com.ibm.rmi.corba.InitialNamingClient.cachedInitialReferences(InitialNamingClient.j
             at com.ibm.rmi.corba.InitialNamingClient.resolve_initial_references(InitialNamingClien
             at com.ibm.rmi.corba.InitialReferenceClient.resolve_initial_references(InitialReferenc
             at com.ibm.rmi.corba.ORB.resolve_initial_references(ORB.java:3211)
             at com.ibm.rmi.iiop.ORB.resolve_initial_references(ORB.java:523)
             at com.ibm.CORBA.iiop.ORB.resolve_initial_references(ORB.java:2898)
             ..........
```

## IBM ORB service: collecting data

This section describes how to collect data about ORB problems.

### 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 30, "JIT diagnostics," on page 295).

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 -fullversion`.
- 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.

# 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.



**Note:** The term "*nix" refers to operating systems that are versions of Unix and Unix-like operating systems, such as AIX, Linux, or MVS.

## 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 *nix 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.

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.

## The font.properties file

The JVM has a font.properties file that controls how Java adds fonts to its runtime. This is platform specific.

### The *nix font.properties file

The font.properties file consists of several sections. The first section associates java font names to platform fonts.

On *nix platforms, a typical font.properties entry looks like this:

```
serif.3=-monotype-timesnewromanwt-medium-r-normal--*-%d-75-75-p-*-ibm-udcjp
```

This can be interpreted as follows:

```
<General font name>.[<Style>.]<index>=<Platform font name>
```

where:
`General font name` is the font name that Java understands.
`Style` can be normal, italic, bold, bolditalic, and so on. The default is ″normal″.
`Index` specifies the sequence of searching for matching font glyphs, with zero the highest priority.
`Platform font name` is the name of the font in the system.

Other entries can be:

**Font name /font file mapping**
Entries in the font.properties help Java to map the font name with the font file

**filename.timesnewromanwt_medium_r=tnrwt_j.ttf**
This shows that the system font timesnewromanwt is defined in the font file tnrwt_j.ttf.

**Font substitution**

When the font is missing, try to map the missing font with another:

```
substitute.0=-timesnewromanwt=timesnewromanwt30
```

Here, if timesnewromanwt font is not found in the system, it is substituted with timesnewromanwt30.

If the JVM cannot load any fonts from the system, the characters are displayed as small squares.

**Font CharSet**
These entries control the converter to be used to convert unicode strings.

**fontcharset.serif.0=sun.iof.CharToByteISO8859_1**
This indicates that to draw the font that is specified by serif.0, the sun.iof.CharToByteISO8859_1 converter is used.

**Alias**
This is used to map one Java font to another.

**alias.timesnewroman=serif**
The font definitions for serif are used for the font timesnewroman.

**Fontset**
The fontset entry is used to match fonts specifically for TextArea and TextField objects.

```
fontset.serif.plain=\
-jdk-lucidabright-medium-r-normal--*-%d-75-75-p-*-iso8859-1,\
-monotype-timesnewromanwt-medium-r-normal--*-%d-75-75-*-*-jisx0208.1983-0,\
-monotype-timesnewromanwt-medium-r-normal--*-%d-75-75-*-*-jisx0201.1976-0,\
-monotype-timesnewromanwt-medium-r-normal--*-%d-75-75-p-*-ibm-udcjp
```

## The Windows font.properties file

Modification of this file is risky and is not supported. See http://java.sun.com/products/jdk/1.2/docs/guide/internat/fontprop.html for more information.

**Note:** The Windows font.properties file refers to Arial Unicode MS. The Arial Unicode MS font is part of Office 2000 and above. You can download it from Microsoft if you have a license for Microsoft Office or related products.

# Font utilities

## Font utilities in *nix platforms

**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--*-%d-75-75-m-*-ibm-udcjp` brings up a window with all the glyphs that are in that font.

**xlsfonts**
Use **xlsfonts** to check whether a particular font is installed on the system. For example: `xlsfonts | grep ksc` will list all the Korean fonts in the system.

**iconv**
Use to convert the character encoding from one encoding to other. Converted text is written to standard output. For example: `iconv -f oldset -t newset [file ...]`

Options are:

**-f oldset**
Specifies the source codeset (encoding).

**-t newset**
Specifies the destination codeset (encoding).

**file**
The file that contain the characters to be converted; if no file is specified, standard input is used.

## Font utilities on Windows systems

There are no built-in utilities similar to those offered by *nix.

## 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 from font.properties; then look into the fontpath to check for the existence of the font. 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 `xlfd <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 (*nix). 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.

# Chapter 22. AS/400 problem determination

The JTC does not provide or support the AS/400® JVM. The AS/400 JVM is provided and serviced by the AS/400 product teams. You can get more information about AS/400 Java from these sites:

http://www-1.ibm.com/servers/eserver/iseries/software/websphere/wsappserver/docs/ws50perfcon.pdf
http://publib.boulder.ibm.com/pubs/pdfs/as400/V4R5PDF/as4ppcp6.pdf (Chapter 7. Java performance guide)
http://ca-web.rchland.ibm.com/perform/websphere/TuningGC.pdf (garbage collection paper)

For more information about the i-Series platform, contact the i-Series team through links at the above sites.

# Chapter 23. OS/2 problem determination

IBM supports the OS/2® Warp® platform at the JVM 1.3.1 level. No more versions of the JVM for OS/2 are planned.

The Diagnostics Guide for V1.3.1 contains information for debugging OS/2.

# Part 4. Using diagnostic tools

This part of the book describes how to use the diagnostic tools that are available. The chapters are:

# Chapter 24. Overview of the available diagnostics

This chapter describes the diagnostic tools used during problem determination. The purpose of this chapter is to describe what is available, with a broad look at how and when you might use a particular tool.

Note that Java on any given platform comprises two parts:

- The Java Virtual Machine (JVM) that interfaces Java to the native operating system and
- The Java classes that provide the infrastructure.

There are no tools that can ″cross the barrier″ between these two parts. In other words, if you have a Java problem you need a Java diagnostic tool and if you have a problem in the JVM you need a JVM diagnostic tool.

This book addresses the IBM JVM. The diagnostics in this book are all JVM diagnostics.

## Categorizing the problem

Problems are considered to fall into four categories:

1. Crashes
2. Hangs
3. Memory leaks
4. Poor performance

You need different tools to solve problems in each category. Most of the tools described in this book are from IBM, either built into the Java Virtual Machine (JVM) or as external monitoring tools.

## Platforms

IBM provides and supports Java on a number of platforms. These platforms can be divided into five groups:

1. Linux
2. Windows
3. AIX (Power PC)
4. z/OS (previously called S/390)
5. Sun Solaris (IBM services the Sun Solaris JVM only when running IBM middleware; for example, a WebSphere application on the Sun Solaris platform).

The platform architectures are very different. You will find that:

- Some tools exist only for a given platform.
- Some tools have different versions for different platforms.
- Some tools are cross-platform.

# Third-party tools

This book refers to third-party tools. Refer to Chapter 38, “Using third-party tools,” on page 383 for more details. This book only outlines whether or not a tool applies to a problem, and refers you to the vendor of that tool for further documentation

# Summary of cross-platform tools

IBM has several cross-platform diagnostic tools. They apply to the different types of problem described above. The following sections provide a brief description of the tools and indicate the sort of problem determination to which they are suited.

## Javadump (or Javacore)

On some platforms, and in some cases, Javadump is known as ″Javacore″.

The code that creates Javadumps is part of the JVM. You can control it by using environment variables and runtime switches. By default, a Javadump is produced when the JVM terminates unexpectedly (crashes) because of an operating system signal or when the user enters a reserved key-combination (for example, Ctrl-Break on Windows). A Javadump is a text file that attempts to summarize the state of the JVM at the instant the signal occurred.

Although Javadump (or Javacore) is present in Sun Solaris JVMs, much of the content of the Javadump is IBM value-add; that is, it is present only in IBM JVMs. See Chapter 25, “Using Javadump,” on page 219 for details. Javadump is an automatic tool; it is simple to use.

## Heapdump

Heapdump is an IBM JVM utility that generates a record of all the Java objects in the Java heap. The Heapdump tool can generate Heapdump files at the request of the user, in an out-of-memory condition, or when the JVM terminates unexpectedly (a crash). Each Heapdump file contains details of every object in the heap at the time it was generated. This is useful for diagnosing several kinds of problems, in particular, memory-related problems.

Heapdump is an IBM value-add tool; that is, it is present only in IBM JVMs. See Chapter 26, “Using Heapdump,” on page 245 for details.

## Cross-platform dump formatter

The cross-platform 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. Platform code to extract data from the dump generated by the native operating system
2. A Java tool to analyze that data

The formatter understands the JVM and can be used to analyze its internals. Thus, it is a useful tool to debug JVM crashes. You must have a basic knowledge of the JVM internals to use this tool. The formatter is really for use on postmortem dumps. However, it is also useful for checking if leak problems occur in JVM resources.

For more information, see Chapter 29, “Using the dump formatter,” on page 261.

The cross-platform dump formatter is an IBM value-add tool; that is, it is present only in IBM JVMs. See Chapter 29, “Using the dump formatter,” on page 261 for details. You need a long time to master the dump formatter; it is not a simple tool to use. However, it is the deepest and most complete post-mortem analysis tool that is available.

## JVMPI tools

JVMPI tools conform to the JVM Profiling Interface that is common across all JVMs. The IBM JVM is fully JVMPI compatible. Any tool conforming to JVMPI can be used to profile the IBM JVM.

JVMPI tools help with problems involving leaks and performance, although profile logs might give useful hints to the state of the JVM just before a crash or hang problem.

The JVMPI is intended for interested parties to write profilers, but IBM provides a useful agent with the IBM SDK.

For more information, see Chapter 36, “Using the JVMPI,” on page 369.

Note that JVMPI is officially described by Sun as “an experimental interface for profiling”. It is not yet a standard profiling interface. It is provided for the benefit of tools vendors who have an immediate need for profiling hooks in the Java virtual machine. Sun states that the JVMPI will continue to evolve, based on feedback from customers and tools vendors. IBM fully supports the current JVMPI specification and is fully compatible with the current Sun release of the technology. Visit Sun’s website (java.sun.com/j2se/1.3/docs/guide/jvmpi) for more information.

## JVMDI tools

JVM Debug Interface (JVMDI) tools are part of the Java Platform Debugging Architecture, which is a common standard for JVMs. The IBM JVM is fully JPDA compatible.

Any JPDA debugger can be attached to the IBM JVM. Being debuggers, these tools are best suited to tracing leaks or the conditions prior to a crash or hang, if these are repeatable.

An example of such a tool is the debugger that is bundled with Eclipse for Java.

## JVM trace

JVM trace is a key diagnostic tool for the JVM.

The IBM JVM contains a large amount of embedded trace. Naturally, this tracing is switched off by default. Command-line options allow you to turn trace on, set exactly what is to be traced, and specify where the trace output is to go.

Trace applies to performance and leak problem determination, although the trace file might provide clues to the state of a JVM before a crash or hang.

Trace is an IBM value-add tool; that is, it is present only in IBM JVMs. See Chapter 33, “Tracing Java applications and the JVM,” on page 321 for details. You need some considerable effort to master trace. However, it is an extremely effective tool.

## JVMRI

The JVM RAS interface is sometimes referred to as JVMRAS. (RAS stands for Reliability, Availability, Serviceability.) This interface allows you to control several JVM operations programmatically.

For example, the IBM JVM contains a large amount of embedded trace. Tracing is switched off by default. A JVMRI agent acts as a plug-in to allow real-time control of trace information. You use the **-Xrun** command-line option so that the agent is loaded by the JVM itself at startup time. When loaded, a JVMRI agent can dynamically switch JVM trace on and off, control the trace level, and capture the trace output. The JVMRI applies to performance and leak problem determination, although the trace file might provide clues to the state of a JVM before a crash or hang.

The RAS plug-in interface is an IBM value-add interface; that is, it is present only in IBM JVMs. See Chapter 35, “Using the Reliability, Availability, and Serviceability interface,” on page 355 for details. You need some programming skills and tools to be able to use this interface.

## JVMMI

The JVM Monitoring Interface is accessed by library code loaded by the JVM. As with JVMRI, you usually control the loading of a JVMMI agent by using the **-Xrun** command-line option, but you can load it also from a JNI program.

This interface allows an external plug-in to request notification of certain events, including, but not limited to, thread start and stop, heap low on memory and full, and class loading and unloading. The interface also allows enumeration, in real time, over particular objects (for example monitors).

The JVM Monitoring Interface can be used as an aid in tracking down problems in performance and memory leak detection. If JVMMI is running at the time of a crash or a hang, it might also be of assistance.

The JVMMI interface is an IBM value-add interface; that is, it is present only in IBM JVMs. For a sample agent and for more information, see Chapter 34, “Using the JVM monitoring interface (JVMMI),” on page 343. You need some programming skills and tools to be able to use this interface.

## Application trace

Application trace allows you to place tracepoints in Java code to provide trace data that is combined with other forms of trace. You can control the tracepoints at start-up or enable them dynamically. For more information, see Chapter 33, “Tracing Java applications and the JVM,” on page 321.

Application trace is an IBM value-add tool; that is, it is present only in IBM JVMs. See Chapter 33, “Tracing Java applications and the JVM,” on page 321 for details. You need some considerable effort to master trace. However, it is an extremely effective tool.

## Method trace

Method trace permits the tracing of Java methods using the existing JVM trace facility. The trace has entry, exit, and, optionally, input parameters. You can select classes and methods for trace using wildcards. You start method trace by command-line options at JVM startup time, or by using a JVMRI agent.

Method trace is an IBM value-add tool; that is, it is present only in IBM JVMs. See Chapter 28, "Using method trace," on page 257 for details. Basic method trace is simple to use, and very effective.

### JVM command line parameters

The IBM JVM has a rich set of command-line parameters that allow you to control various functions. See Appendix G, "Command-line parameters," on page 487.

### JVM environment variables

The IBM JVM has a rich set of environment variables that you can use to affect its running; for example, controlling the JIT.

The variables are separately described for the tools and diagnostics to which they apply, and are also all gathered together for reference in Appendix E, "Environment variables," on page 407

## Platform tools

Platform-specific tools are documented in the appropriate sections that follow. All platforms (except z/OS) have a dump extractor tool that feeds the cross-platform dump formatter. For the other tools, each platform has a different toolset. Some tools have versions for two or more platforms.

The Java service team has a prototype Java application that displays and analyses the Java environment variables. If you want more details about this prototype, send an e-mail to jvmcookbook@uk.ibm.com.

# Chapter 25. 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, native stack, locks, and memory. The exact contents are dependent 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.

**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 220
- "Interpreting a Javadump" on page 221

**Note:** "Interpreting a Javadump" on page 221 is the main part of this chapter.

## Enabling a Javadump

Javadumps are enabled by default. To turn them off, set the environment variable **DISABLE_JAVADUMP** to any value. The **DISABLE_JAVADUMP** environment variable is not available on z/OS. For more information, see "z/OS environment variables" on page 411.

You can use the **JAVA_DUMP_OPTS** environment variable to control exactly when a Javadump is produced; see Chapter 27, "JVM dump initiation," on page 251 for more information.

## The location of the generated Javadump

The JVM checks each of the following locations for existence and write-permission, and stores the Javadump in the first one available. Note that 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).
2. The current working directory of the JVM processes.
3. The location specified by the **TMPDIR** environment variable, if set.
4. The /tmp directory or, on Windows only, the location specified by the **TEMP** environment variable, if set.
5. Windows only: If the Javadump cannot be stored in any of the above, it is put to STDERR.

### location of generated Javadump

On Linux and AIX a log of Javadump files is maintained in the file /tmp/javacore_locations.

The file name is of the following form: (where PID is the process ID and TIME is the number of seconds since 1/1/1970.)

*Table 9. Javadump filename formats*

| Platform | Javadump filename format |
|---|---|
| Windows and Linux | javacore.YYYYMMDD.HHMMSS.PID.txt |
| AIX | javacorePID.TIME.txt |
| z/OS | JAVADUMP.YYYYMMDD.HHMMSS.PID.txt |

## Triggering a Javadump

The Javadump is generated when one of the following occurs:

- **A fatal native exception** occurs in the JVM (not a Java Exception).
- **The JVM has completely run out of heapspace**.

  **Note:** You can disable this option by setting the **IBM_JAVADUMP_OUTOFMEMORY=FALSE** environment variable.
- **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 the **JAVA_DUMP_OPTS** environment variable. For example, you can optionally get a Javadump when the JVM terminates normally (on an interrupt). See Chapter 27, "JVM dump initiation," on page 251 for more information.

A "fatal" exception is one that will cause the JVM to terminate. The JVM handles this by producing a Javadump and then returning control to the operating system. The behavior of the JVM in a failure is not affected by the Javadump and should not affect the production of core files. However, it is possible that the processing that is done to generate a Javadump might itself find a problem. In this unlikely event, switch off Javadumps with the **DISABLE_JAVADUMP=TRUE** environment variable.

**Note:** The exact format and content might be different to what is documented at this stage.

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.

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 is subject to the same Javadump environment variables as are described in "Enabling a Javadump" on page 219.

You can get a Javadump in a "totally out of heapspace" condition; that is, at the same time as the Java application receives an OutOfMemory error. This feature is enabled by default. You can disable it by using the **IBM_JAVADUMP_OUTOFMEMORY=FALSE**) environment variable.

## Interpreting a Javadump

The information that is provided in a Javadump file depends on the platform on which you are running the JVM.

**Notes:**

1. In some conditions information might be missing because of the nature of a crash.
2. Most of the cross-platform sections of the dump are fully documented in the Windows example. The Linux, AIX, and z/OS examples build on top of the Windows sections to describe platform specifics.

### Javadump tags

The Javadump files contain tags. This metadata makes it easier to parse and perform simple analysis on the contents of Javadump files. An example tag is:

```
1CIJAVAVERSION J2RE 1.4.1 IBM Windows 32 build cn141-20030601
```

Normal tags have these characteristics:

- Tags are up to 15 characters long (padded with spaces).
- The first digit is a nesting level.
- The second and third characters identify the component that wrote the message (for example, CI, DG, LK).
- The remainder is a unique string.

Special tags have these characteristics:

- A tag of "NULL" means the line is purely to aid readability
- Every section is headed by a tag of "0SECTION" with the section title

Here is an example of some tags:

```
NULL           ------------------------------------------------------------------------
0SECTION       TITLE subcomponent dump routine
NULL           ===============================
1TISIGINFO     signal 24 received
1TIDATETIME    Date:                 2003/05/21 at 11:49:02
1TIFILENAME    Javacore filename:    /u/riccole/JAVADUMP.20030521.114902.50332001.txt
NULL           ------------------------------------------------------------------------
0SECTION       XHPI subcomponent dump routine
NULL           ===============================
1XHSIGRECV     SIGQUIT received at 0 in (Default handler)
1XHTIME        Wed May 21 11:49:02 2002
1XHFULLVERSION Java J2RE 1.4.1 IBM OS/390 Persistent Reusable VM build cm141-20030521
NULL
1XHOPENV       Operating Environment
NULL           ---------------------
2XHNOTCPIPINFO Could not get TCPIP information for host WINMVS16
2XHOSLEVEL     OS Level         : z/OS V01 R02.00 Machine 9672 Node MV16
2XHCPUS        Processors -
```

```
3XHCPUARCH       Architecture   : (not implemented)
3XHNUMCPUS       How Many       : (not implemented)
3XHCPUSENABLED   Enabled        : 5
NULL
```

**Note:** For the rest of the chapter, the tags are removed to aid readability.

## Locks, monitors, and deadlocks (LK)

Here is an example of the LK component part of the dump (this is practically the same on all platforms). The LK component handles locking in the JVM.

A lock 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.

A monitor is a special kind of locking mechanism that is used in the JVM to allow flexible synchronization between threads. For the purpose of this section, read the terms monitor and lock interchangeably.

To avoid having a monitor on every object, the JVM usually uses a flag in a class or method block to indicate that the item is locked. Most of the time, a piece of code will transit some locked section without contention. Therefore, the guardian flag is enough to protect this piece of code. This is called a flat monitor. However, if another thread wants to access some code that is locked, a genuine contention has occurred. The JVM must now create (or inflate) the monitor object to hold the second thread and arrange for a signaling mechanism to coordinate access to the code section. This monitor is now called an inflated monitor.

```
 ------------------------------------------------------------------------
LK subcomponent dump routine
============================

Monitor pool info:
  Initial monitor count: 32
  Minimum number of free monitors before expansion: 5
  Pool will next be expanded by: 16
  Current total number of monitors: 32
  Current number of free monitors: 28

Monitor Pool Dump (flat & inflated object-monitors):
  sys_mon_t:0x3020D9E8 infl_mon_t: 0x3020D4D8:
    java.lang.ref.Reference$Lock@302D20C0/302D20C8: <unowned>
       Waiting to be notified:
          "Reference Handler" (0x3477F210)
  sys_mon_t:0x3020DA68 infl_mon_t: 0x3020D500:
    java.lang.ref.ReferenceQueue$Lock@302D1CD0/302D1CD8: <unowned>
       Waiting to be notified:
          "Finalizer" (0x34784B20)
  sys_mon_t:0x3020DAE8 infl_mon_t: 0x00000000:
    java.lang.Object@3030DF38/3030DF40: Flat locked by thread ident 0x07, entry count 1
       Waiting to be notified:
          "Thread-0" (0x353366F0)
  sys_mon_t:0x3020DB68 infl_mon_t: 0x00000000:
    java.lang.Object@3030DF48/3030DF50: Flat locked by thread ident 0x06, entry count 1
       Waiting to be notified:
          "Thread-1" (0x35336E90)

JVM System Monitor Dump (registered monitors):
    Integer lock access-lock: <unowned>
    Evacuation Region lock: <unowned>
    Heap Promotion lock: <unowned>
```

```
    Sleep lock: <unowned>
    Method trace lock: <unowned>
    Heap lock: owner "Signal dispatcher" (0x3477AFD0), entry count 1
    Monitor Cache lock: owner "Signal dispatcher" (0x3477AFD0), entry count 1
    JNI Pinning lock: <unowned>
    JNI Global Reference lock: <unowned>
    Classloader lock: <unowned>
    Binclass lock: <unowned>
    Monitor Registry lock: owner "Signal dispatcher" (0x3477AFD0), entry count 1
    Thread queue lock: owner "Signal dispatcher" (0x3477AFD0), entry count 1
       Waiting to be notified:
          "Thread-2" (0x30210B00)

Thread identifiers (as used in flat monitors):
    ident 0x02 "Thread-2" (0x30210B00) ee 0x302108D4
    ident 0x07 "Thread-1" (0x35336E90) ee 0x35336C64
    ident 0x06 "Thread-0" (0x353366F0) ee 0x353364C4
    ident 0x05 "Finalizer" (0x34784B20) ee 0x347848F4
    ident 0x04 "Reference Handler" (0x3477F210) ee 0x3477EFE4
    ident 0x03 "Signal dispatcher" (0x3477AFD0) ee 0x3477ADA4

Java Object Monitor Dump (flat & inflated object-monitors):
    java.lang.ref.ReferenceQueue$Lock@302D1CD0/302D1CD8
        locknflags 80000200 Monitor inflated infl_mon 0x3020D500
    java.lang.ref.Reference$Lock@302D20C0/302D20C8
        locknflags 80000100 Monitor inflated infl_mon 0x3020D4D8
    java.lang.Object@3030DF38/3030DF40
        locknflags 00070000 Flat locked by thread ident 0x07, entry count 1
    java.lang.Object@3030DF48/3030DF50
        locknflags 00060000 Flat locked by thread ident 0x06, entry count 1
```

The component dump is split into the following sections:

- **Monitor pool info:** This keeps track of the state of the LK component.
- **Monitor Pool Dump (flat & inflated object-monitors):** These are the objects that threads are waiting to lock.

Consider the monitor that is described by the part:

```
sys_mon_t:0x3020DAE8 infl_mon_t: 0x00000000:
  java.lang.Object@3030DF38/3030DF40: Flat locked by thread ident 0x07, entry count 1
     Waiting to be notified:
         Thread-0" (0x353366F0)
```

The first line gives the address of some JVM monitor structures. The second line shows that a lock is on the `java.lang.Object@3030DF38/3030DF40`. Object and thread number 0x07 has this lock. The entry count 1 says that one thread is inside a method or block that is protected by the lock. The fourth line shows that a thread called Thread-0 with its JVM sys_thread_t structure at 0x353366F0 is waiting for the lock.

Note that it is possible for the entry count to be higher than 1 because a method could use a wait() call in a synchronized method to release a lock temporarily so that another thread could then take the lock. In this case, two threads would be in synchronized methods or blocks, but only one would actually have the lock at any one time.

## JVM system monitor dump (registered monitors)

This is a list of monitors that are maintained for use by the JVM. Each lock contains details of which thread (including their respective JVM sys_thread_t data structure addresses) holds the lock, if applicable.

### Thread identifiers (as used in flat monitors)

This section contains a list of the threads. This line describes thread number 0x07 called Thread-1 with JVM data structure at address 0x35336E90:

```
ident 0x07 "Thread-1" (0x35336E90) ee 0x35336C64
```

### Java object monitor dump (flat & inflated object-monitors)

This section is similar to "Monitor Pool Dump (flat & inflated object-monitors)", but with some additional JVM internal information.

### Using the LK component dump to diagnose a deadlock

Deadlocks are usually caused by an inconsistency in the locking semantics of the application, or possibly some aspect of the JVM. This leads to one of the following conditions:

- Thread 1 has lock A and wants lock B
- Thread 2 has lock B and wants lock A

That is: Thread 1 waits for B is locked by Thread 2 waits for A is locked by Thread 1..... - a cycle in the "waits for/locked by" graph.

Neither thread can proceed until the other releases the relevant lock; this cannot happen. This might be more complex, involving three or more threads with interdependent locks, but the principle remains the same. Other threads usually end up blocked on one or other of the locks involved, thereby causing a totally deadlocked Java application.

```
sys_mon_t:0x3020DAE8 infl_mon_t: 0x00000000:
              (1)
  java.lang.Object@3030DF38/3030DF40: Flat locked by thread ident 0x07, entry count 1
     Waiting to be notified:
        "Thread-0" (0x353366F0)
sys_mon_t:0x3020DB68 infl_mon_t: 0x00000000:
              (2)
  java.lang.Object@3030DF48/3030DF50: Flat locked by thread ident 0x06, entry count 1
     Waiting to be notified:
        "Thread-1" (0x35336E90)
```

The above LK component dump is an example of a deadlock. First notice (see (1) above ) that Thread-0 is waiting to lock the `java.lang.Object@3030DF38/3030DF40` object. Now thread number 0x07 has the lock of this object. Looking at the Thread numbers section, this is the thread called Thread-1. Conversely, Thread-1 is waiting to lock `java.lang.Object@3030DF48/3030DF50`, which is held by thread number 0x06 - Thread-0. This is a clear (and in this case simple) deadlock.

Javadump can automatically diagnose most deadlocks. Here is an example:

```
Deadlock detected !!!
---------------------

Thread "Thread-1" (0x35336E90)
  is waiting for:
    sys_mon_t:0x3020DB68 infl_mon_t: 0x00000000:
    java.lang.Object@3030DF48/3030DF50:
  which is owned by:
Thread "Thread-0" (0x353366F0)
  which is waiting for:
    sys_mon_t:0x3020DAE8 infl_mon_t: 0x00000000:
    java.lang.Object@3030DF38/3030DF40:
  which is owned by:
Thread "Thread-1" (0x35336E90)
```

Similarly, the dump formatter (see Chapter 29, "Using the dump formatter," on page 261) can also diagnose deadlocks.

**Note:** Some categories of deadlock cannot be diagnosed automatically; they require understanding of the synchronization in the application. For example, if threads have interdependencies on wait()/notify() operations, you cannot be aware, from the diagnostic information, of which thread would be expected to notify some thread that is waiting.

## Javadump sample output 1 (Windows)

This is a sample of the top section of a Javadump file that was produced on Windows.

The following Javadump sample output 1 (Windows) section applies to Linux and AIX also.

```
------------------------------------------------------------------------
TITLE subcomponent dump routine
==============================
signal 11 received
Date:                 2004/05/21 at 14:54:38
Javacore filename:    C:\javacore.20040521.145438.2008.txt
------------------------------------------------------------------------
XHPI subcomponent dump routine
==============================
Exception code: C0000005 Access Violation
Fault address:  00F51090 01:00000090
Fault module:   c:\sdk\jre\bin\classic\jvm.dll

Registers:
EAX:00000000
EBX:0006FE98
ECX:0006FFB0
EDX:00000000
ESI:00000002
EDI:007F7A50
CS:EIP:001B:00F51090
SS:ESP:0023:0006FE58  EBP:0006FE60
DS:0023  ES:0023  FS:0038  GS:0000

Flags:00010202

------------------------------------------------------------------------
CI subcomponent dump routine
============================
J2RE 1.4.2 IBM Windows 32 build cn142-20040521
Running as a standalone JVM
java -Xmx6m -Xbootclasspath/p:D:\fix.jar -classpath classes Pause
Java Home Dir:   c:\sdk\jre
Java DLL Dir:    c:\sdk\jre\bin
Sys Classpath:   c:\sdk\jre\lib\core.jar;c:\sdk\jre\lib\server.jar;
UserArgs:
  -Dconsole.encoding=Cp850
  vfprintf 0x402E00
  -XrunagentGPF
  -Dinvokedviajava
  -Djava.class.path=classes
  vfprintf

JVM Monitoring Interface (JVMMI)
--------------------------
No events are enabled.
...
...
```

```
...
------------------------------------------------------------------------
XM subcomponent dump routine
============================

Exception Info
--------------
JVM Exception 0x2 (subcode 0x0) occurred in thread "main" (TID:0x92B9B8)
```

## File header (TITLE) - signal information

The top of the file shows general information about the dump. In this case, a signal 11 (a SIGSEGV) occurred in the JVM and caused it to crash. Looking down the dump at the XM section, you can see that it occurred in a thread called "main" with TID (Thread Identifier) 0x92B9B8). This can be cross-referenced against other parts of the file; for example, the stack traces and locks.

## XHPI section

This section is platform-specific. It contains the information that you get for a crash on Windows. It lists the cause of the problem and where it happened (an access violation in jvm.dll at address 00F51090). It also shows which registers were in use at the time of the problem.

## System properties (CI)

This section of the file shows:

- SDK Version and Build Identifier: `J2RE 1.4.2 IBM Windows 32 build cn142-20040521`.
- The command-line argument that started the JVM: `java -Xmx6m -Xbootclasspath/p:D:\fix.jar -classpath classes Pause`.
- The location from which the Runtime Environment executables were loaded: `c:\sdk\jre\bin`.
- The default bootclasspath: `Sys Classpath: c:\sdk\jre\lib\core.jar;c:\sdk\jre\lib\server.jar;`.
- Arguments supplied when initializing the JVM labeled under `UserArgs`. For example: `-Djava.class.path=classes`.
- Events that are tracked by the JVM Monitoring Interface (none in this case).

The main diagnostic function of this section is to determine exactly what native executables and Java classes were being run when the dump occurred. This can include the **java** executable, the Java API, IBM extensions, and user application class files.

The `UserArgs` section shows arguments for the JVM, which might have been supplied by the user or generated during JVM initialization. For example, the `-Djava.class.path=classes` property was generated by the user specifying the option `-classpath classes` on the command line. However, `-Dconsole.encoding=Cp850` was generated automatically.

The bootclasspath (classpath of the bootstrap class loader) contains the locations from which the Java API is loaded. This takes the value listed under "Sys Classpath" and is then modified by any supplied arguments. In this case, the `-Xbootclasspath/p:D:\fix.jar` argument adds `D:\fix.jar` to the start of the default bootclasspath. The effect of this is that the JVM will attempt to load classes (including the Java API) from `D:\fix.jar` before `c:\sdk\jre\lib\core.jar;c:\sdk\jre\lib\server.jar`.

The classpath (generally refers to the system or application class loader's classpath) takes a default value of "." (the working directory of Java). This causes the `-Djava.class.path=.` property to be set by default. In this example, it is then overridden by `-Djava.class.path=classes` (generated from the command-line option `-classpath classes`). This example shows the case where a later value of a system property replaces an earlier value in the system property list.

## Data Conversion (DC)

The DC component dump gives information on internal functions used to convert various character formats and data used to handle Java types.

```
------------------------------------------------------------------------
DC subcomponent dump routine
===========================
Header eye catcher  DCST
Header length       24
Header version      1
Header modification 0
DC Interface at 0x60ED78 with 15 entries
   1 - dcCString2JavaString           0x4A56EA
   2 - dcInt642CString                0x4A5938
   3 - dcJavaString2NewCString        0x4A5BF2
   4 - dcJavaString2CString           0x4A5ACF
   5 - dcJavaString2NewPlatformString 0x4A5D67
   6 - dcJavaString2UTF               0x4A6253
   7 - dcPlatformString2JavaString    0x4A5E79
   8 - dcUnicode2UTF                  0x4A607D
   9 - dcUnicode2UTFLength            0x4A5F9B
  10 - dcUTF2JavaString               0x4A6800
  11 - dcUTFClassName2JavaString      0x4A6B6D
  12 - dcJavaString2ClassName         0x4A6372
  13 - dcUTF2UnicodeNext              0x4A6E22
  14 - dcVerifyUTF8                   0x4A701E
  15 - dcDumpRoutine                  0x4A7A20
Array info at 0x5A98B8 with 16 entries
   1 - index  0 signature 0 name      N/A factor 0
   2 - index  0 signature 0 name      N/A factor 0
   3 - index  2 signature L name  class[] factor 4
   4 - index  0 signature 0 name      N/A factor 0
   5 - index  4 signature Z name   bool[] factor 1
   6 - index  5 signature C name   char[] factor 2
   7 - index  6 signature F name  float[] factor 4
   8 - index  7 signature D name double[] factor 8
   9 - index  8 signature B name   byte[] factor 1
  10 - index  9 signature S name  short[] factor 2
  11 - index 10 signature I name    int[] factor 4
  12 - index 11 signature J name   long[] factor 8
  13 - index  0 signature 0 name   uint[] factor 0
  14 - index  0 signature 0 name  uint1[] factor 0
  15 - index  0 signature 0 name  uint2[] factor 0
  16 - index  0 signature 0 name  uint3[] factor 0
```

## Diagnostics settings (DG)

This part of the dump output gives information about the size of the buffer used to hold the Javadump before being flushed to disk and information about JVM trace settings.

```
------------------------------------------------------------------------
DG subcomponent dump routine
===========================
Trace enabled: Yes
  Trace activated
  Trace: Internal
Javadump buffer size (allocated): 2621440
```

## Storage Management (ST)

See Chapter 2, "Understanding the Garbage Collector," on page 7 for information about how the ST component works. This part of the file gives various storage management values, including:

- Whether concurrent mark is used (Concurrent GC: No)
- Current heap address limits (between 102601fc and 1065fbfc)
- Counts of allocation failures (AF Counter: 0) and garbage collection cycles (GC Counter: 0)
- Free space in heap (2e42c0) and size of current heap (3ffa00)

```
------------------------------------------------------------------------
ST subcomponent dump routine
===========================
Resettable GC: No
Concurrent GC: No
Current Heap Base: 102601fc
Current Heap Limit: 1065fbfc
Middleware Heap Base: 102601fc
Middleware Heap Limit: 1065fbfc
Number of GC Helper Threads: 0
-Xconcurrentlevel: 0
-Xconcurrentbackground: 0
GC Counter: 0
AF Counter: 0
Bytes of Heap Space Free: 2e42c0
Bytes of Heap Space Allocated: 3ffa00
SM Base: 0
SM End: 0
PAM Start: 0
PAM End: 0
Compact Action: 0
```

## Execution Engine (XE)

This part of the dump contains information such as JIT initialization and the JIT mixed-mode compilation threshold (2000).

```
------------------------------------------------------------------------
XE subcomponent dump routine
===========================
MMI threshold for Java methods is set to 2000
JIT is initialized
JVMPI is not activated
MMI threshold for JNI methods is set to 0
Trace history length is set to 4
```

## Threads and stack trace (XM)

This section shows a complete list of Java threads that are alive.

```
XM subcomponent dump routine
============================

Current Thread Details
----------------------
    "Finalizer"(TID:0x901900,sys_thread_t:0x8818D0,state:CW,native ID:0x734)prio=8
        at java.lang.Object.wait(Native Method)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:133)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:148)
        at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:213)

All Thread Details
------------------
Full thread dump Classic VM (J2RE 1.4.2 IBM build cn1420-20040608,native threads):
    "Thread-1"(TID:0x9017A0,sys_thread_t:0x23EAC8,state:R,native ID:0x6E4)prio=5
    "Thread-0"(TID:0x9017E8,stillborn,sys_thread_t:0x112360D0,state:R,native ID:0x7C0)prio=5
    "Finalizer"(TID:0x901900,sys_thread_t:0x8818D0,state:CW,native ID:0x734)prio=8
        at java.lang.Object.wait(Native Method)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:133)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:148)
        at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:213)
    "Reference Handler"(TID:0x901948,sys_thread_t:0x87F660,state:CW,native ID:0x748)prio=10
        at java.lang.Object.wait(Native Method)
        at java.lang.Object.wait(Object.java:429)
        at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:131)
    "Signal dispatcher"(TID:0x901990,sys_thread_t:0x870A28,state:R,native ID:0x7B4)prio=5
```

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:

```
"Thread-1" (TID:0x9017A0, sys_thread_t:0x23EAC8, state:R, native ID:0x6E4) prio=5
```

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.

Below each thread there is a stack trace for that thread. A stack trace is a representation of the hierarchy of Java method calls made by the thread. For example:

```
 "Reference Handler"(TID:0x901948,sys_thread_t:0x87F660,state:CW,native ID:0x748)prio=10
        at java.lang.Object.wait(Native Method)
        at java.lang.Object.wait(Object.java:429)
        at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:131)
```

The java.lang.ref.Reference$ReferenceHandler.run() method calls java.lang.Object.wait() which calls java.lang.Object.wait() which is then waiting on some condition (thread state is CW). To the right of each method name called is the source of the code for the method. Examples of this are:

- `at java.lang.Object.wait(Object.java:429)` - The `wait` method is at line 429 of a Java source file called Reference.java
- `at java.lang.Object.wait(Object.java(Compiled Code))` - The `wait()` method is executing JIT-compiled code from a Java source file called Reference.java. See “Refining a stack trace using the JIT options (XM)” on page 230.

- at java.lang.Object.wait(Native Method) - The wait() method is a native method. This could be user application JNI code or (as in this case) Java API implementation.

## Refining a stack trace using the JIT options (XM)

Chapter 30, "JIT diagnostics," on page 295 explains how to use the JIT options.

In a stack trace such as:

```
at count.output(count.java(Compiled Code))
at count.loop(count.java(Compiled Code))
at count.main(count.java:4)
```

You can see that the main method has been compiled by the JIT (Just-In-Time) compiler. This dynamically compiles Java bytecode into native code for greater execution efficiency. However, because of the optimization for speed, the source code line numbers are not maintained. If you want to trace the execution path, you might want to know where loop() called output(). In this case, loop() is compiled and so no source code line number is given. There might be only one place in loop() where output() is called. If not, you could disable the JIT or increase the JIT threshold and rerun the application. For example, with the JIT disabled, the stack trace now becomes:

```
at count.output(count.java:15)
at count.loop(count.java:10)
at count.main(count.java:4)
```

Now the exact path through to output() is available. It is also now possible to see where execution in the output() method was at the point of the Javadump (line 15), which could be useful in a crash or hang situation. Note that in most cases the program behavior will be the same with the JIT on or off (however, events will occur at different speeds). However, in some cases there could be a JIT problem or the user Java application might have a race condition.

In this example, the trace shows that inline methods might not appear.

```
"Thread-0" (TID:0x102B3FA0, sys_thread_t:0x27CF18, state:R, native ID:0x2FA4) prio=5
   Stack trace (In-lined methods may not appear)
        at lot.run()V
```

This is another example where turning the JIT off (or disabling inlining) can help to get more information about a program path.

Consider the case when a JIT problem occurs, and, for example, the Javadump is produced when the JVM crashes. Turning the JIT off might prevent the crash and therefore the Javadump that is being produced.

In the case of a Java application that has a race condition, the behavior of the program depends on the speed at which different parts execute. So turning the JIT on or off could also affect whether a crash or a hang occurs at all.

Turning off the JIT might affect the path that the application takes to get to the point of the Javadump. Because of this, you then might see a different stack trace from the original. At this point, you could try skipping only the compiling method in which you were interested, by setting the environment variable export JITC_COMPILEOPT=SKIP{count}{*}. In the example above, skipping compiling method output could produce a stack trace like:

```
at count.output(count.java:15)
at count.loop(count.java(Compiled Code))
at count.main(count.java:4)
```

This enables you to see where method `output()` is in execution, leaving the JIT turned on for all the other methods.

## Classloaders and Classes (CL)

See Chapter 3, "Understanding the class loader," on page 31 for information about the parent-delegation model. The classloader section includes:

- Classloader summaries. The defined class loaders and the relationship between them
- Classoader loaded classes. The classes that are loaded by each classloader

In this example, there are the standard three classloaders:

- Application classloader (sun/misc/Launcher$AppClassLoader), which is a child of the
- Extension classloader (sun/misc/Launcher$ExtClassLoader), which is a child of the
- Bootstrap (sometimes called "system") classloader (*System*).

As an example, take the application classloader with the full name sun/misc/Launcher$AppClassLoader. Under ″Classloader summaries″, this has flags `-----ta-`, which, from the key above, shows that the class loader is `6=trusted` and `7=application`. It gives the number of loaded classes (1) and the parent classloader sun/misc/Launcher$ExtClassLoader(0x102B4098). The parent address 0x102B4098 corresponds to the entry Shadow 0x102B4098 for the extension classloader entry below. Under the ″ClassLoader loaded classes″ heading, you can see that the application classloader has loaded one class called `lot` at address 0x102B0110.

```
------------------------------------------------------------------------
CL subcomponent dump routine
============================
 Classpath Z(c:\sdk\jre\lib\core.jar),Z(c:\sdk\jre\lib\server.jar)
 Oldjava mode false
 Bootstrapping false
 Verbose class dependencies false
 Class verification VERIFY_REMOTE
 Namespace to classloader 0x00000000
 Start of cache entry pool 0x00000000
 Start of free cache entries 0x20BF362C
 Location of method table 0x002521C0
 Global namespace anchor 0x00B308F4
 System classloader shadow 0x00249830
 Classloader shadows 0x0026BC90
 Extension loader 0x102B4098
 System classloader 0x102B4030
 Classloader summaries
         12345678: 1=primordial,2=extension,3=shareable,4=middleware,5=system,6=trusted,
    7=application,8=delegating
         -----ta- Loader sun/misc/Launcher$AppClassLoader(0x0026BC90), Shadow 0x102B4030,
    Parent sun/misc/Launcher$ExtClassLoader(0x102B4098)
                 Number of loaded classes 1
                 Number of cached classes 177
                 Allocation used for loaded classes 1
                 Package owner 0x102B4030
         -xh-st-- Loader sun/misc/Launcher$ExtClassLoader(0x00269C90), Shadow 0x102B4098,
            Parent *none*(0x00000000)
                 Number of loaded classes 2
                 Number of cached classes 8
                 Allocation used for loaded classes 3
                 Package owner 0x102B4098
         p-h-st-- Loader *System*(0x00249830), Shadow 0x00000000
                 Number of loaded classes 249
                 Number of cached classes 249
```

```
                Allocation used for loaded classes 3
                Package owner 0x00000000
 ClassLoader loaded classes
        Loader sun/misc/Launcher$AppClassLoader(0x0026BC90)
                lot(0x102B0110)
        Loader sun/misc/Launcher$ExtClassLoader(0x00269C90)
                com/ibm/crypto/provider/IBMJCA(0x20C50218)
                com/ibm/crypto/provider/IBMJCA$1(0x20C50318)
        Loader *System*(0x00249830)
                java/lang/Character(0x00B55118)
                java/io/OutputStream(0x00B58118)
                java/util/Collection(0x00B51018)
.....................left out to save space.....................
                java/io/ObjectStreamField(0x00B50718)
```

### Final section

This section gives the Javadump buffer size and usage and gives confirmation that the file is complete.

```
------------------------------------------------------------------------
Javadump End section
Javadump Buffer Usage Information
================================
Javadump buffer size (allocated): 2621440
Javadump buffer size (used)     : 27752
---------------------- END OF DUMP -----------------------------------
```

## Javadump sample output 2 (Linux)

This section describes the Linux-specific parts of the Javadump; the cross-platform sections are covered above in the Windows section and should be read before this section.

The following is the top section of a Javadump file that was produced on Linux.

```
------------------------------------------------------------------------
TITLE subcomponent dump routine
==============================
signal 3 received
Date:                2003/05/30 at 13:08:05
Javacore filename:   /home/dave/code/dump/javacore868.1034946485.txt
------------------------------------------------------------------------
XHPI subcomponent dump routine
==============================
Fri May 30 13:08:05 2003
SIGQUIT received in <unknown> at (nil) in <unknown>.
J2RE 1.4.1 IBM build cxia32141-20030530
Operating Environment
---------------------
Host  : golkonda.
OS Level : 2.4.10-4GB.#1 Fri Sep 28 17:20:21 GMT 2001
glibc Version : 2.2.4
Processors -
  Architecture     : (not implemented)
  How Many         : (not implemented)
  Enabled          : 1
Memory Info
-----------        total:    used:    free:  shared: buffers:  cached:
Mem:  525729792 505630720 20099072        0 48328704 207196160
Swap: 271392768  4218880 267173888
MemTotal:       513408 kB
MemFree:         19628 kB
MemShared:           0 kB
Buffers:         47196 kB
Cached:         198220 kB
SwapCached:       4120 kB
Active:         235400 kB
Inactive:        14136 kB
HighTotal:           0 kB
HighFree:            0 kB
LowTotal:       513408 kB
LowFree:         19628 kB
SwapTotal:      265032 kB
SwapFree:       260912 kB
User Limits (in bytes except for NOFILE and NPROC) -
-----------
 RLIMIT_FSIZE : infinity
 RLIMIT_DATA : infinity
 RLIMIT_STACK : 2093056
 RLIMIT_CORE : 0
 RLIMIT_NOFILE : 1024
 RLIMIT_NPROC : 4094
Signal Handlers
---------------
HUP            : intrDispatchMD (libhpi.so)
INT            : intrDispatchMD (libhpi.so)
QUIT           : intrDispatchMD (libhpi.so)
ILL            : intrDispatchMD (libhpi.so)
TRAP           : intrDispatchMD (libhpi.so)
ABRT           : intrDispatchMD (libhpi.so)
FPE            : intrDispatchMD (libhpi.so)
KILL           : default handler
BUS            : intrDispatchMD (libhpi.so)
SEGV           : intrDispatchMD (libhpi.so)
```

```
PIPE           : ignored
ALRM           : default handler
USR1           : sigusr1Handler (libhpi.so)
USR2           : get_self (libhpi.so)
TERM           : intrDispatchMD (libhpi.so)
CLD            : default handler
Environment Variables
--------------------
PWD=/home/dave/code/frame
.....................(left out to save space).....................
PATH=/home/dave/jdks/141SR1/bin:/home/dave/bin:/usr/local/bin:/usr/bin:/usr/X11R6
  /bin:/bin:/usr/lib/java/bin:/usr/games/bin:/usr/games:/opt/gnome/bin:/opt/kde2/bin:.:/opt/cmvc/bin
LC_COLLATE=POSIX
IBM_JAVA_COMMAND_LINE=/java/jre/bin/exe/java -Djava.compiler=NONE frame
```

### File header (TITLE) and XHPI header - signal information

At the top of the file the following information is given:

- Date and time when the Javadump was produced.
- The signal that caused the dump to be produced. In this case, `SIGQUIT` (indicating a user-initiated Javadump). This line also gives the function and library in which the signal occurred. This is `<unknown>at (nil)in <unknown>` in this case because the signal was sent by the user to the JVM to initiate the dump
- The Java version and build identifier.

### XHPI - operating environment

This section gives information about the Linux environment that Java is running on. The information includes:

- The host name (or machine name). In this example: *golkonda*.
- The Linux kernel level. In this example: *2.4.10-4GB*.
- The glibc version. In this example:*2.2.4*.
- The number of processors. In this example: *1*.

### XHPI - memory information

This part of the XHPI section contains various virtual memory statistics. In this example: 19628/513408 KB of physical memory is free and 260912/265032 KB of swap space is free.

### XHPI - user limits

This part of the XHPI section contains various user environment limits (as reported by **ulimit**). For example:

- RLIMIT_FSIZE - maximum size of any files created
- RLIMIT_ DATA - maximum data segment size
- RLIMIT_STACK - maximum stack size
- RLIMIT_CORE - maximum core file size
- RLIMIT_NOFILE - maximum number of open files
- RLIMIT_NPROC - maximum number of processes

### XHPI - signal handlers

This part of the XHPI section describes the functions that are installed to handle various signals on behalf of the JVM - for example, `intrDispatchMD` in `libhpi.so` handles most operating system signals.

### XHPI - environment variables

This part of the XHPI section describes environment variables set within the Java process. For example, these include:

- The working directory (**PWD**)
- The path (**PATH**)
- The command line used to invoke Java (stored in **IBM_JAVA_COMMAND_LINE** by the JVM)

## XHPI - memory map

The location of the memory map of the JVM processes is given in the Javadump file. The memory map is held in the /proc/<PID>/maps file. Use the memory map to identify:

- The names of particular system, Java or user libraries and executables that were loaded.
- The locations in the filesystem that these libraries and executables were loaded from.
- The contents of various memory addresses. For example, you might have an exception address that you want to identify.

Each row represents a memory region. The columns are as follows:

```
ADDRESS-RANGE   PERMS  OFFSET  DEV   INODE       PATHNAME
```

**ADDRESS-RANGE**
The address space that the region occupies

**PERMS** Permissions:
- **r** = read
- **w** = write
- **x** = execute
- **s** = shared
- **p** = private (copy on write)

**OFFSET** The offset into file or entity from which the region is mapped.

**DEV** The major and minor device numbers from where the file or entity comes.

**INODE** The inode of the file or entity. The value is 0 if no file or entity is associated with region.

**PATHNAME**
The pathname of file or entity (if applicable).

Each of the native libraries here has two memory regions. The first is the text segment and the second is the data segment, which is reflected in the permissions (text - read and execute, data - read and write). For the most recent information about memory maps and the proc file system see the section 5 man page for proc: `man 5 proc`.

```
Memory map
----------

08048000-0804c000 r-xp 00000000 00:0d 3153929    /java/jre/bin/exe/java
0804c000-0804d000 rw-p 00003000 00:0d 3153929    /java/jre/bin/exe/java
0804d000-081a8000 rwxp 00000000 00:00 0
10000000-1fab0000 rwxp 00000000 00:00 0
40000000-40014000 r-xp 00000000 08:02 346279     /lib/ld-2.2.4.so
40014000-40015000 rw-p 00013000 08:02 346279     /lib/ld-2.2.4.so
40015000-40016000 rw-p 00000000 00:00 0
40016000-4001b000 r-xp 00000000 00:0d 3145740    /java/jre/bin/libxhpi.so
4001b000-4001c000 rw-p 00004000 00:0d 3145740    /java/jre/bin/libxhpi.so
4001c000-4001d000 r--p 00000000 08:02 1389931    /usr/lib/locale/en_GB/LC_IDENTIFICATION
4001d000-4001e000 r--p 00000000 08:02 981135     /usr/lib/locale/en_GB/LC_MEASUREMENT
4001e000-4001f000 r--p 00000000 08:02 1389937    /usr/lib/locale/en_GB/LC_TELEPHONE
```

```
4001f000-40020000 r--p 00000000 08:02 327046     /usr/lib/locale/en_GB/LC_ADDRESS
40020000-40021000 r--p 00000000 08:02 327064     /usr/lib/locale/en_GB/LC_NAME
40021000-40022000 r--p 00000000 08:02 981143     /usr/lib/locale/en_GB/LC_PAPER
40022000-40023000 r--p 00000000 08:02 1196712    /usr/lib/locale/en_GB/LC_MESSAGES/SYS_LC_MESSAGES
40023000-40031000 r-xp 00000000 08:02 343429     /lib/libpthread.so.0
40031000-40039000 rw-p 0000d000 08:02 343429     /lib/libpthread.so.0
40039000-4004b000 r-xp 00000000 08:02 343420     /lib/libnsl.so.1
4004b000-4004d000 rw-p 00011000 08:02 343420     /lib/libnsl.so.1
4004d000-4004f000 rw-p 00000000 00:00 and 0
4004f000-40051000 r-xp 00000000 08:02 343417     /lib/libdl.so.2
40051000-40053000 rw-p 00001000 08:02 343417     /lib/libdl.so.2
40053000-4016e000 r-xp 00000000 08:02 343413     /lib/libc.so.6
4016e000-40175000 rw-p 0011a000 08:02 343413     /lib/libc.so.6
40175000-40179000 rw-p 00000000 00:00 0
40179000-402fb000 r-xp 00000000 00:0d 3153921    /java/jre/bin/classic/libjvm.so
402fb000-40301000 rw-p 00181000 00:0d 3153921    /java/jre/bin/classic/libjvm.so
40301000-40314000 rw-p 00000000 00:00 0
40314000-40335000 r-xp 00000000 08:02 343418     /lib/libm.so.6
40335000-40336000 rw-p 00020000 08:02 343418     /lib/libm.so.6
40336000-40348000 r-xp 00000000 00:0d 3145739    /java/jre/bin/libhpi.so
40348000-4034a000 rw-p 00011000 00:0d 3145739    /java/jre/bin/libhpi.so
4034a000-4036d000 rw-p 00000000 00:00 0
4036d000-40bc4000 rwxp 00000000 00:00 0
40bc4000-40be7000 r-xp 00000000 00:0d 3145741    /java/jre/bin/libjava.so
40be7000-40beb000 rw-p 00022000 00:0d 3145741    /java/jre/bin/libjava.so
40beb000-40bec000 rw-p 00000000 00:00 0
40bec000-40bfa000 r-xp 00000000 00:0d 3145742    /java/jre/bin/libzip.so
40bfa000-40bfd000 rw-p 0000d000 00:0d 3145742    /java/jre/bin/libzip.so
40bfd000-413b6000 r--s 00000000 00:0d 2375908    /java/jre/lib/rt.jar
413b6000-413e7000 rw-p 00000000 00:00 0
413e7000-41764000 r--s 00000000 00:0d 2375882    /java/jre/lib/i18n.jar
41764000-4178f000 r--p 00000000 08:02 981132     /usr/lib/locale/en_GB/LC_CTYPE
4178f000-41790000 r--p 00000000 08:02 866679     /usr/lib/locale/en_GB/LC_MONETARY
41790000-41796000 r--p 00000000 08:02 719511     /usr/lib/locale/en_GB/LC_COLLATE
41796000-41797000 r--p 00000000 08:02 719506     /usr/lib/locale/en_GB/LC_TIME
41797000-41798000 r--p 00000000 08:02 981141     /usr/lib/locale/en_GB/LC_NUMERIC
41798000-41799000 r--p 00000000 08:02 196231     /usr/share/locale/en_GB/LC_MESSAGES/libc.mo
41799000-4179b000 r-xp 00000000 08:02 1978713    /usr/lib/gconv/ISO8859-1.so
4179b000-4179c000 rw-p 00001000 08:02 1978713    /usr/lib/gconv/ISO8859-1.so
4179c000-417f9000 r--s 00000000 00:0d 2375819    /java/jre/lib/ext/ibmjcaprovider.jar
417f9000-41804000 r--s 00000000 00:0d 2375820    /java/jre/lib/ext/indicim.jar
41804000-41886000 rw-p 00000000 00:00 0
......................(left out to save space).....................
bffc9000-c0000000 rwxp 0000a000 00:00 0
```

This provides useful information such as:

- The locale being used (usr/lib/locale/en_GB in this case)
- Where Java was loaded from (the SDK was in /java)
- Which Java native libraries were loaded (for example, libjvm.so, libxhpi.so, libhpi.so, libjava.so)
- The threading library that is being used (/lib/libpthread.so.0)
- The glibc being used (/lib/libc.so.6)
- Java system libraries and API libraries (for example, rt.jar, i18n.jar, ibmjcaprovider.jar, indicim.jar)
- The area in memory where the Java heap is stored (in most cases it will be the largest region starting at 0x10000000 in this case fab0000 in size which is about 250 MB)

An examination of the memory map helps you to highlight problems such as incorrect system libraries being loaded from the wrong location or with the wrong filename or version.

```
Current Thread Details
----------------------
      Native Stack of ""Signal dispatcher"" PID 11898
      ----------------------------------
 GetHeapDumpFileName_Impl at 4001929F in libxhpi.so
 Diagnostics_Impl at 40019409 in libxhpi.so
 dgGenerateJavacore at 40249F2E in libjvm.so
 xmSetProtectionDomain at 402DD220 in libjvm.so
 xmExecuteThread at 402DF2E1 in libjvm.so
 double2ll at 402D94EE in libjvm.so
 sysThreadRegs at 40343B0A in libhpi.so
pthread_detach at 40029F37 in libpthread.so.0
 __clone at 4012ABAA in libc.so.6
------------------------------------------------------------------------
```

## System properties (CI)

This section is the same as the Windows one and is covered in "System properties (CI)" on page 226.

## Data conversion (DC)

This section is the same as the Windows one and is covered in "Data Conversion (DC)" on page 227.

## Diagnostics settings (DG)

This section is the same as the Windows one and is covered in "Diagnostics settings (DG)" on page 227.

## Storage management (ST)

This section is the same as the Windows one and is covered in "Storage Management (ST)" on page 228.

## Execution engine (XE)

This section is the same as the Windows one and is covered in "Execution Engine (XE)" on page 228.

## Locks, monitors, and deadlocks (LK)

For details of using the LK component dump to diagnose deadlocks see "Locks, monitors, and deadlocks (LK)" on page 222.

## Threads and stack trace (XM)

See "Threads and stack trace (XM)" on page 228 for a description of a Java Stack trace.

The Linux version of the XM component dump is similar to the Windows one except that it contains additional native stack information. See the dump below for an example of the additions.

The top stack trace (taken from the bottom of the XHPI section of the dump) is for the thread that dealt with the signal that caused the Javadump. This is called the "Signal dispatcher" thread with PID 11898. This thread is also listed among the other threads further down the dump output.

Each stack frame visible contains the name, address, and library of the function involved (this information is not always complete). For example, the function `dgGenerateJavacore` at address `40249F2E` from library libjvm.so was the one that generated the Javadump.

Debug builds are also available from IBM. These include line numbers for frames within JVM libraries.

```
XM subcomponent dump routine
============================

Exception Info
--------------
Not available

Thread Info
-----------
Full thread dump Classic VM (J2RE 1.4.1 IBM build cxia32141-20030530, native threads):
    "Thread-2" (TID:0x10051758, sys_thread_t:0x819C3F8, state:CW, native ID:0x1406) prio=5
 at frame.run(frame.java:52)
 at java.lang.Thread.run(Thread.java:512)
  PID: 11902
  ----- Register Values -----
REG_EAX : FFFFFFFC, REG_EBX : BEFFF804, REG_ECX : 8, REG_EDX : 1F, REG_ESI : BEFFF804,
 REG_EDI : BEFFF804, REG_EBP : BEFFF7D4
---------------------------------------------------------
  PID: 11902
  ----- Native Stack -----
 pthread_getconcurrency at 0x4002c7f0 in libpthread.so.0
 pthread_cond_wait at 0x40028f3d in libpthread.so.0
 condvarWait at 0x4033e6da in libhpi.so
 sysMonitorWait at 0x40340d22 in libhpi.so
 lkMonitorEnter at 0x40260737 in libjvm.so
 mmipInvokeSynchronizedJavaMethodWithCatch at 0x402d14c7 in libjvm.so
 iiq_doinvoke_V__ at 0x402aadfe in libjvm.so
 EJinvq_doinvoke_V__ at 0x402a5013 in libjvm.so
 ??
---------------------------------------------------------
.....................(left out to save space).....................
---------------------------------------------------------
    "Signal dispatcher" (TID:0x10051990, sys_thread_t:0x80D6008, state:R, native ID:0x402) prio=5
  PID: 11898
  ----- Register Values -----
REG_EAX : 0, REG_EBX : 80D6008, REG_ECX : 1, REG_EDX : 403493C0, REG_ESI : 4
, REG_EDI : 80D6008, REG_EBP : BF7FF9E8
---------------------------------------------------------
  PID: 11898
  ----- Native Stack -----
 ??
---------------------------------------------------------
 PID: 11898
  ----- Native Stack -----
 ??
---------------------------------------------------------
    "main" (TID:0x100519D8, sys_thread_t:0x8058C30, state:CW, native ID:0x400) prio=5
 at java.lang.Object.wait(Native Method)
 at java.lang.Thread.join(Thread.java:958)
 at java.lang.Thread.join(Thread.java:1011)
 at frame.main(frame.java:21)
  PID: 11883
  ----- Register Values -----
REG_EAX : FFFFFFFC, REG_EBX : BFFFF3DC, REG_ECX : 8, REG_EDX : BFFFF38C, REG_ESI
: BFFFF3DC, REG_EDI : BFFFF3DC, REG_EBP : BFFFF3AC
---------------------------------------------------------
  PID: 11883
  ----- Native Stack -----
 pthread_getconcurrency at 0x4002c7f0 in libpthread.so.0
 pthread_cond_wait at 0x40028f3d in libpthread.so.0
 condvarWait at 0x4033e6da in libhpi.so
 sysMonitorWait at 0x40340d22 in libhpi.so
 lkMonitorWait at 0x40261104 in libjvm.so
 JVM_MonitorWait at 0x4021b49e in libjvm.so
 mmipSysInvokeJni at 0x402d1078 in libjvm.so
 mmisInvokeJniMethodHelper at 0x402d0bf7 in libjvm.so
 mmipInvokeJniMethod at 0x402d16b3 in libjvm.so
```

```
ivq_doinvoke_V__ at 0x402aa96b in libjvm.so
ivq_doinvoke_V__ at 0x402aa96b in libjvm.so
EJisq_doinvoke_V__ at 0x402a512f in libjvm.so
??
------------------------------------------------------------
```

Below each Java stack trace (for example, see "main" above) there are the hardware register values, the PID of the process that implements the thread, and the native stack.

### Refining a stack trace using the JIT options (XM)

See "Refining a stack trace using the JIT options (XM)" on page 230.

### Classloaders and classes (CL)

This section is the same as the Windows one and is covered in "Classloaders and Classes (CL)" on page 231.

### Final section

This section is the same as the Windows one and is covered in "Final section" on page 232.

## Javadump sample output 3 (AIX)

This section describes the AIX-specific parts of the Javadump; the cross-platform sections are covered above in the Windows section, and should be read before this section.

### File header (TITLE) and XHPI header - signal information

This section is almost identical to the corresponding one on Linux, as described in "File header (TITLE) and XHPI header - signal information" on page 234 except that details of the "Signal dispatcher" thread and stack trace are given at the top of the file instead of at the top of the XM component.

```
Fri May 30 13:07:20 2003
SIGQUIT received at 0x0 in <unknown>.
J2RE 1.4.1 IBM AIX build ca141-20030530
Current Thread Details
----------------------
    "Signal dispatcher" sys_thread_t:0x3436E538
  ----- Native Stack -----
 unavailable - iar 0x0 not in text area
------------------------------------------------------------------------
```

### XHPI - operating environment

See"XHPI - operating environment" on page 234 for a brief description of the contents of this section.

```
Operating Environment
---------------------
Host   : jtc170-43.hursley.ibm.com:9.20.178.89
OS Level  : AIX 4.3.3.0
Processors -
 Architecture : POWER_PC (impl: POWER_630, ver: PV_630)
 How Many : 1
 Enabled  : 1
```

### XHPI - user limits

See"XHPI - operating environment" on page 234 for a brief description of the contents of this section.

```
User Limits (in bytes except for NOFILE and NPROC) -
 RLIMIT_FSIZE : 1073741312
 RLIMIT_DATA : 2147483645
```

```
 RLIMIT_STACK : 33554432
 RLIMIT_CORE : 1073741312
 RLIMIT_NOFILE : 2000
 NPROC(max) : 262144
Page Space (in blocks) -
 /dev/hd6: size=524288, free=523883
```

### XHPI - signal handlers

This section is identical to that on Linux and is covered in "XHPI - signal handlers" on page 234.

### XHPI - environment variables

This section is identical to that on Linux and is covered in "XHPI - environment variables" on page 234.

### XHPI - loaded libraries

The AIX Javadump has a section on loaded libraries instead of the memory map that is displayed in the Linux Javadump. Here is a sample output:

```
Loaded Libraries (sizes in bytes)
---------------------------------
/srvbuild/ca131-20020810/inst.images/rios_aix_4/sdk/jre/bin/libjitc.a
 filesize : 2690612
 text start : 0xD1621000
 text size : 0x22DA90
 data start : 0x3458DD40
 data size : 0xC304
/usr/lib/libiconv.a
 filesize : 377832
 text start : 0xD0034100
 text size : 0x13F3A
 data start : 0x3452EDA0
 data size : 0xA574
/usr/lib/libi18n.a
 filesize : 123742
 text start : 0xD002C100
 text size : 0x7ADC
 data start : 0x3453A6D0
 data size : 0x112C
.....................(left out to save space).....................
/usr/lib/libc.a
 filesize : 6424258
 text start : 0xD145D720
 text size : 0x1C358F
 data start : 0xF06C98A0
 data size : 0x7F968
```

For each loaded library, the pathname, filesize, and address ranges for text and data segments are given. See the "XHPI - memory map" on page 235, because the basic principles of interpretation are the same.

### System properties (CI)

This section is the same as for Windows, and is described in "System properties (CI)" on page 226.

### Data conversion (DC)

This section is the same as for Windows, and is described in "Data Conversion (DC)" on page 227.

### Diagnostics settings (DG)

This section is the same as for Windows, and is described in "Diagnostics settings (DG)" on page 227.

### Storage management (ST)
This section is the same as for Windows, and is described in “Storage Management (ST)” on page 228.

### Execution engine (XE)
This section is the same as for Windows, and is described in “Execution Engine (XE)” on page 228.

### Locks, monitors, and deadlocks (LK)
For details of how to use the LK component dump to diagnose deadlocks, see “Locks, monitors, and deadlocks (LK)” on page 222.

### Threads and stack trace (XM)
This section is very similar to the related Linux section, and is described in “Threads and stack trace (XM)” on page 237.

### Refining a stack trace using the JIT options (XM)
This section is the same as for Windows, and is described in “Refining a stack trace using the JIT options (XM)” on page 230.

### Classloaders and classes (CL)
This section is the same as for Windows, and is described in “Classloaders and Classes (CL)” on page 231.

### Final section
This section is the same as for Windows, and is described in “Final section” on page 232.

## Javadump sample output 4 (z/OS)
The z/OS Javadump format is mainly comprised of cross-platform sections that are the same as in the Windows Javadump. Some other sections are very similar to those on AIX and Linux. This section includes some sample output where it differs from Windows, AIX, or Linux.

### File header (TITLE) and XHPI header - signal information
This section is the same as for Linux, and is described in “File header (TITLE) and XHPI header - signal information” on page 234.

### XHPI - operating environment
This section is the same as for Linux, and is described in “XHPI - operating environment” on page 234.

### XHPI - user limits
This section is the same as for Linux, and is described in “XHPI - user limits” on page 239.

### XHPI - signal handlers
This section is the same as for Linux, and is described in “XHPI - signal handlers” on page 234.

### XHPI - environment variables
This section is the same as for Linux, and is described in “XHPI - environment variables” on page 234.

### XHPI - loaded libraries

This section is the same as for AIX, and is described in "XHPI - loaded libraries" on page 240.

### XHPI - thread counts

This section contains quantitative information about threads.

```
Thread Counts
-------------
Total  Thread Count: 5
Active Thread Count: 5
JNI    Thread Count: (not implemented)
```

### System properties (CI)

This section is the same as for Windows, and is described in "System properties (CI)" on page 226.

### Data conversion (DC)

This section is the same as for Windows, and is described in "Data Conversion (DC)" on page 227.

### Diagnostics settings (DG)

This section is the same as for Windows, and is described in "Diagnostics settings (DG)" on page 227.

### Storage management (ST)

This section is the same as for Windows, and is described and is covered in "Storage Management (ST)" on page 228.

### Execution engine (XE)

This section is the same as for Windows, and is described in "Execution Engine (XE)" on page 228.

### Locks, monitors, and deadlocks (LK)

For details of how to use the LK component dump to diagnose deadlocks, see "Locks, monitors, and deadlocks (LK)" on page 222.

### Threads and stack trace (XM)

Here is an example of the stack trace that is given on z/OS:

```
Current Thread Details
----------------------
"Signal dispatcher" (sys_thread_t:176ab400)
      Native Thread State: SYSTEM RUNNING
      Native Stack Data: base: 17129660 top: 0 pointer: 171a9660 used(e8ed69a0) free(171a9660)
      Monitors Held: (not implemented)

      Native Stack
      ------------

Program name                                Entry Name  Statement ID
      /u/sovbld/hm131s/hm131s-20020923/src/xhpi/pfm/xhpi.c
Diagnostics  1889
      /jtc/riccole/cm131s/src/jvm/sov/dg/dg_javacore.c
dgGenerateJavacore   495
      /u/sovbld/hm131s/hm131s-20020923/src/jvm/sov/xm/signals.c
signalDispatcherThread   338
      /u/sovbld/hm131s/hm131s-20020923/src/jvm/sov/xm/thr.c
xmExecuteThread  1450
      /u/sovbld/hm131s/hm131s-20020923/src/jvm/pfm/xe/common/xe_thread_md.c threadStart    79
      /u/sovbld/hm131s/hm131s-20020923/src/hpi/pfm/threads_utils.c
```

```
ThreadUtils_Shell   900
                                                @@GETFN
      CEEOPCMM                                  CEEOPCMM
```

The details above for the current thread are taken from the bottom of the XHPI section of the dump. The remainder of the information below is under the XM section of the dump.

```
XM subcomponent dump routine
============================

Exception Info
--------------
JVM Exception 0x4 (subcode 0x1) occurred in thread "Signal dispatcher" (TID:0x18004190)

Native stack at exception generation:
Program Name                                Entry Name                      Statement ID
/u/sovbld/hm131s/hm131s-20020923/src/hpi/pfm/exception_md.c
                                            sysSignalCatchHandler           335
/u/sovbld/hm131s/hm131s-20020923/src/hpi/pfm/interrupt_md.c
                                            userSignalHandler               397
/u/sovbld/hm131s/hm131s-20020923/src/hpi/pfm/interrupt_md.c
                                            intrDispatch                    657
                                            @@GETFN
                                            __zerros
CEEHDSP                                     CEEHDSP
/jtc/riccole/cm131s/src/jvm/sov/dg/dg_javacore.c
                                            dgGenerateJavacore
/u/sovbld/hm131s/hm131s-20020923/src/jvm/sov/xm/signals.c
                                            signalDispatcherThread          338
/u/sovbld/hm131s/hm131s-20020923/src/jvm/sov/xm/thr.c
                                            xmExecuteThread                 1450
/u/sovbld/hm131s/hm131s-20020923/src/jvm/pfm/xe/common/xe_thread_md.c
                                            threadStart                     79
/u/sovbld/hm131s/hm131s-20020923/src/hpi/pfm/threads_utils.c
                                            ThreadUtils_Shell               900
                                            @@GETFN
CEEOPCMM                                    CEEOPCMM
```

Interpreting this information is very similar to doing so with a Linux Javadump. See “Threads and stack trace (XM)” on page 237 for more details.

## Refining a stack trace using the JIT options (XM)

This section is the same as for Windows, and is described in “Refining a stack trace using the JIT options (XM)” on page 230.

## Classloaders and classes (CL)

This section is the same as for Windows, and is described in “Classloaders and Classes (CL)” on page 231.

## Final section

This section is the same as for Windows, and is described in “Final section” on page 232.

# Chapter 26. Using Heapdump

This chapter describes:



## Information for users of previous releases of Heapdump

Heapdumps in release 1.4.1, Service Refresh 1 and above of the IBM JVM, are different from those of release 1.4.1. These environment variables are now supported as they were in release 1.3.1:

- **IBM_HEAPDUMP**
- **IBM_HEAPDUMP_OUTOFMEMORY**

Heapdumps now always trigger a garbage collection that occurs before the dump. This means that all the objects that are in the Heapdump are live (reachable) objects. Earlier versions of Heapdump could include objects that were not reachable and so were eligible for garbage collection. This change improves the accuracy and effectiveness of the Heapdump mechanism.

## Summary of Heapdump

Heapdump is an IBM JVM facility that generates a dump of all the live objects that are on the Java heap; that is, those that are used by the Java application. This dump is called a Heapdump. It shows the objects that are using large amounts of memory on the Java heap, and what is preventing them from being collected by the Garbage Collector.

The Heapdump contains two lines per object. The first line displays the address of the object, various flags (see "Sample Heapdump output" on page 248), its size, and type information. The second line contains a list of the memory addresses of objects that have been referenced by that object.

## Enabling a Heapdump

You can generate a Heapdump in either of two ways:

- Explicit generation
- JVM-triggered generation

When the Javaheap is exhausted, JVM-triggered generation is enabled by default, as are Heapdumps that are generated by other programming methods. To enable

signal-based Heapdumps, you must set the IBM_HEAPDUMP=TRUE environmental variable or the appropriate JAVA_DUMP_OPTS before you start the Java process.

**Note:** If you disable all signal-based dumps (Javadumps and System dumps) that use JAVA_DUMP_OPTS (see Chapter 27, "JVM dump initiation," on page 251), all signal-based Heapdumps are disabled.

To disable generation of a Heapdump, on platforms other than Windows use:

```
unset IBM_HEAPDUMP
unset  IBM_HEAP_DUMP
```

On Windows, use:

```
set IBM_HEAPDUMP=
set IBM_HEAP_DUMP=
```

## Explicit generation of a Heapdump

You can explicitly *generate* a Heapdump in either of the following ways:

- By sending a signal to the JVM from the operating system
- By using the HeapDump() method inside Java code that is being executed
- By using the JVMRI to request a Heapdump from a loaded agent

You can explicitly *request* a Heapdump in the same way as you can a Javadump. Before the Heapdump starts, the heap is locked and remains locked until the whole Heapdump file is written to disk. This operation can affect the behavior of your Java application, and make it unresponsive while the dump is being produced.

For Linux and AIX, send the JVM the signal SIGQUIT (kill -3, or Ctrl+\ in the console window).

For Windows, generate a SIGINT (press the Ctrl+Break keys simultaneously).

You can explicitly request a Heapdump from a Java method. The class com.ibm.jvm.Dump contains a static HeapDump() method that causes Java code to initiate a Heapdump, provided that the IBM_HEAPDUMP environment variable is set.

## Triggered generation of a Heapdump

The following events automatically trigger the JVM to produce a Heapdump:

- A fatal native exception occurs in the JVM (not a Java Exception)
- An OutOfMemory or heap exhaustion condition occurs (optional)

If Heapdumps are enabled, they are normally produced immediately before a Javadump. They are produced also if the JVM terminates unexpectedly (a crash).

You can also generate a Heapdump when the Java heap has become full.

This option is enabled by default, and gives a snapshot of the Java heap when no more memory is available. Usually, this snapshot is the most useful output to help you determine the cause of an OutOfMemory condition that is related to the Java heap. This works independently of IBM_HEAPDUMP. So, by default, you get Heapdumps only when no more heap space is available; you do not get Heapdumps in crashes or through a signal to the JVM. You can disable this

feature, and a similar one for Javadumps, by using the IBM_HEAPDUMP_OUTOFMEMORY=FALSE and IBM_JAVADUMP_OUTOFMEMORY=FALSE respectively.

## 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 by the IBM_HEAPDUMPDIR 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 (X:\tmp for Windows, where X is the current working drive)

Note that enough free disk space must be available for the Heapdump file to be written correctly.

On Linux and AIX, a log of Heapdump files is maintained in the file /tmp/javacore_locations. Table 10 shows the format of the Heapdump filename for various platforms.

*Table 10. Format of Heapdump filenames*

| **Platform** | **Heapdump filename format** |
|---|---|
| Windows | heapdump.YYYYMMDD.HHMMSS.PID.txt |
| Linux & AIX | heapdumpPID.TIME.txt |
| z/OS | HEAPDUMP.YYYYMMDD.HHMMSS.PID.txt |
| **Note:** PID is the process ID. TIME is the number of seconds since 1/1/1970 | |

## Producing a compressed Heapdump text file from a System Dump

For information, see “Commands from DvHeapDumpPlugin” on page 274.

## Sample Heapdump output

Heapdumps can be very large with millions of items in them. Here is a small sample:

```
// Version: J2RE 1.4.1 IBM AIX build cadev-20030722
0x30270200 <LF:42 map> [388832] byte[]
0x302cf0e0 <LF:0 map> [304] class HeapConsume

0x302cf210 <LF:42 map> [50232] byte[]
0x302db648 <LF:0 map> [128] sun/misc/Launcher$AppClassLoader
        0x303144b0 0x302db6c8 0x303145c8 0x303147b8 0x30314748 0x3036eb10 0x303146e8
        0x0 0x0 0x30314680 0x0 0x30314610 0x0 0x0 0x30314578 0x30314508 0x30314428 0x303148b8 0x3030c1a8 0x30314490
0x302db6c8 <LF:0 map> [128] sun/misc/Launcher$ExtClassLoader
        0x303151d0 0x0 0x30315360 0x303155c0 0x30315550 0x0 0x303154f0 0x3030c110
        0x0 0x30315488 0x0 0x30315418 0x0 0x0 0x30315310 0x303152a0 0x3031a4a8 0x303155f0 0x3030c1a8 0x303151b0 0x3030be00
0x302db748 <LF:0 map> [96] java/util/logging/LogManager$Cleaner
        0x0 0x0 0x302e8600 0x0 0x302f43f8 0x0 0x0 0x302f1a30 0x0 0x302f4730
0x302db7a8 <LF:0 map> [88] java/lang/Thread
        0x0 0x0 0x302e7748 0x0 0x302eeae8 0x0 0x0 0x302f1a30 0x0
0x302db800 <LF:0 map> [88] java/lang/Thread
        0x0 0x0 0x302e7748 0x0 0x302eeb40 0x0 0x0 0x302f1a30 0x0
0x302db858 <LF:0 map> [88] java/lang/Thread
        0x0 0x0 0x302e7748 0x0 0x302eeb68 0x0 0x0 0x302f1a30 0x0
0x302db8b0 <LF:0 map> [88] java/lang/ref/Finalizer$FinalizerThread
        0x0 0x0 0x302e7748 0x0 0x302eebf8 0x0 0x0 0x302f1a30 0x0
0x302db908 <LF:0 map> [88] java/lang/ref/Reference$ReferenceHandler
        0x0 0x0 0x302e7748 0x0 0x302eed30 0x0 0x0 0x302f1a30 0x0
0x302db960 <LF:0 map> [88] java/lang/Thread
        0x0 0x0 0x302e7748 0x0 0x302eeef8 0x0 0x0 0x302f1a30 0x0
0x302db9b8 <LF:0 map> [88] java/lang/Thread
        0x0 0x302ef750 0x302e8600 0x0 0x302ef010 0x0 0x302db648 0x302f1a30 0x0
0x302db908 <LF:0 map> [88] java/lang/ref/Reference$ReferenceHandler
        0x0 0x0 0x302e7748 0x0 0x302eed30 0x0 0x0 0x302f1a30 0x0
0x302db960 <LF:0 map> [88] java/lang/Thread
        0x0 0x0 0x302e7748 0x0 0x302eeef8 0x0 0x0 0x302f1a30 0x0
0x302db9b8 <LF:0 map> [88] java/lang/Thread
        0x0 0x302ef750 0x302e8600 0x0 0x302ef010 0x0 0x302db648 0x302f1a30 0x0

..............(more data).................


// Breakdown - Classes: 321, Objects: 2624425, ObjectArrays: 486, PrimitiveArrays: 1447
// Meta-data - NullRefs: 2623863, Mark(m): 2626359, Alloc(a): 0, Dosed(d): 13, Pinned(p): 0, MultiPinned(P): 0
// EOF:  Total 'Objects',Refs(null) : 2626679,2629818(1797)
```

The first line in the Heapdump contains the build identifier of the JVM that produced the dump.

In each line of data of the main part of the dump:

- The first hex number is the handle of the object.
- The text in angle brackets gives data about the object:
  - The locknflags word value
  - Whether the object is marked (m)
  - Whether the object has the corresponding alloc bit set (a)
  - Whether the object is dosed (d), pinned (p) or multiply-pinned (P)
- The number in square brackets is the size of the object.
- The remainder of the line is a description of the object. If an object references other objects, those objects are presented as a series of object handles indented in the following line or lines. In the example above, all possible references are dumped, with nulls being explicitly shown as 0x0.

The last three lines in the Heapdump summarize the heap contents.

# Finding memory leaks by using Heapdump

You can use Heapdump to pinpoint memory leaks in both these conditions:
- Out Of Memory exceptions
- Steady memory leaks

## Out Of Memory exceptions

First, ensure that the JVM is set to produce a Heapdump in this condition, as described in "Triggered generation of a Heapdump" on page 246. You can format and inspect the dump to see where unexpectedly large numbers of objects appear. For example, 10 000 objects of type com.ibm.widget.WorkUnit might indicate a problem.

See "Using the HeapRoots post-processor to process Heapdumps" for help in interpreting Heapdumps.

## Steady memory leaks

If your application has a steady memory leak, you can trigger a Heapdump at various points of execution to snapshot the heap at those points. You can then compare the various snapshots to try to identify a problem.

See "Using the HeapRoots post-processor to process Heapdumps" for help in interpreting Heapdumps.

# Using the HeapRoots post-processor to process Heapdumps

HeapRoots is a Heapdump analysis tool that is written in Java.

**Notes:**

1. HeapRoots is an unofficial tool and is provided "as-is".
2. HeapRoots is only a simple tool that can help you to solve memory problems. It does not correct your problem.

The HeapRoots tools can analyze a Heapdump (see the previous sections of this chapter), and can provide analysis that is based on:

- Heap roots
- Objects
- Object types
- Heap gaps (the spaces that are between objects)
- References to and from a given object

Output is provided as an indented tree or as a flat list. All output can be sorted and filtered.

Because HeapRoots is being constantly developed, it is not described in detail here. For the latest information, see http://www.alphaworks.ibm.com/tech/heaproots.

# How to write a JVMMI Heapdump agent

You can write a JVMMI agent to create your own custom Heapdumps.

If the event **JVMMI_EVENT_HEAPDUMP** is enabled, your agent is called to take a Heapdump *instead of* the standard built-in Heapdump.

To find out how to enable this event and see a sample JVMMI Heapdump agent, see Chapter 34, “Using the JVM monitoring interface (JVMMI),” on page 343.

If you decide to write your own Heapdump agent, all the information available in the built-in Heapdump is available.

If you want to use Heaproots to post-process the Heapdump file you create, you must adhere strictly to the format of the Heapdump file described in “Sample Heapdump output” on page 248, with the exception of comment lines – those beginning with // – which are ignored by the Heaproots tool.

## Using VerboseGC to obtain heap information

Use the VerboseGC 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 **-verbosegc** option:

```
java -verbosegc
```

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

# Chapter 27. JVM dump initiation

The JVM supports the ability to generate a native system dump. In addition, a very simple scripting ability allows you to choose when and how a dump is generated. The exact dump created is, by definition, platform dependent. How analysis of the dump proceeds depends upon the platform tools that are available. A short description of how to proceed with the JVM native dump is provided.

## Overview

The JVM might produce dump files in response to specific events, depending on the setting of the environment variables **JAVA_DUMP_OPTS** and **JAVA_DUMP_TOOL**.

These events (or conditions) are grouped as follows:

**EXCEPTION**
Unexpected synchronous terminating signal; that is, unrecoverable storage violation.

**ERROR**
Controlled abort due to an error detected internally; for example, no more memory is available.

**INTERRUPT**
Asynchronous terminating signal; for example, you pressed Ctrl-C.

**DUMP**
This can be caused if you press Ctrl-BREAK on Windows, CtrlL-V on z/OS, or Ctrl-\ on AIX or Linux.

**OUTOFMEMORY**
The JVM cannot satisfy a request for storage.

The types of dump that can be produced (platform specific variations are noted below) are:

1. **SYSDUMP**. An unformatted dump that the operating system generated (basically a core file).
2. **user specified**. Whatever the **JAVA_DUMP_TOOL** variable specifies.
3. **HEAPDUMP**. An internally-generated dump of the objects that are on the Java heap.
4. **JAVADUMP**. An internally-generated and formatted analysis of the JVM.

If all types of dump are requested, they are produced in the above sequence (JAVADUMP always being last). You can read system dumps by using native dump analysis tools (IPCS, dbx, and so on), although they are usually intended as input to the JVM Dump Formatter.

SYSDUMP file names and locations vary with each platform and are detailed below. For more information about JAVADUMP files, see Chapter 25, “Using Javadump,” on page 219.

If any external dump exit routines have been registered, they are run before the main JVM dump sequence (see above), and can optionally terminate all further dump processing by returning RAS_DUMP_ABORT.

## Settings

Which dumps are produced for which condition is determined by the JAVA_DUMP_OPTS variable as follows:

```
JAVA_DUMP_OPTS="ONcondition(dumptype[count],dumptype[count]),ONcondition(dumptype[count],...)"
```

where:

- *condition* can be:
  - ANYSIGNAL
  - DUMP
  - ERROR
  - INTERRUPT
  - EXCEPTION
  - OUTOFMEMORY
- and *dumptype* can be:
  - ALL
  - NONE
  - JAVADUMP
  - SYSDUMP
  - HEAPDUMP
- and *count* is the maximum number of dumps of this type to produce. The count parameter is optional. If the count parameter is not specified, there is no limit to the number of dumps produced.

The default, if **JAVA_DUMP_OPTS** is not set, is:

For platforms other than z/OS:

```
JAVA_DUMP_OPTS="ONANYSIGNAL(JAVADUMP,HEAPDUMP),ONINTERRUPT(NONE)"
```

For z/OS:

```
JAVA_DUMP_OPTS="ONANYSIGNAL(JAVADUMP,SYSDUMP),ONINTERRUPT(NONE)"
```

which indicates that for all conditions, system and Java dumps are to be produced, except for the case of an interrupt condition where no dumps are to be produced. **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 ONANYSIGNAL condition is parsed before all others, so
ONINTERRUPT(NONE),ONANYSIGNAL(SYSDUMP)
has the same effect as
ONANYSIGNAL(SYSDUMP),ONINTERRUPT(NONE).

If **JAVA_DUMP_TOOL** environment variable is set, that variable is assumed to specify a valid executable name and is passed unchanged to the system unless the word %pid is detected in the string. If %pid is detected in the string, the string is replaced with JVM's own process ID, but only for those conditions where a SYSDUMP has been requested. The JVM dump is run after the system dump has been taken, but before anything else.

## Platform-specific variations

Conditions can be mapped to different signals on different platforms, and some signals are recognized on some platforms but not on others. Table 11 shows the mapping across platforms. Note that if the JVM receives a signal that it does not recognize (that is, it is not mapped to a condition as per the table), the default operating system action for that signal is taken; usually the signal is ignored.

*Table 11. Signal mappings on different platforms*

| | z/OS | AIX | Windows | Linux |
|---|---|---|---|---|
| **EXCEPTION** | SIGTRAP | SIGTRAP | | SIGTRAP |
| | SIGILL | SIGILL | SIGILL | SIGILL |
| | SIGSEGV | SIGSEGV | SIGSEGV | SISEGV |
| | SIGFPE | SIGFPE | SIGFPE | SIGFPE |
| | SIGBUS | SIGBUS | | SIGBUS |
| | SIGSYS | SIGSYS | | |
| | SIGXCPU | SIGXCPU | | SIGXCPU |
| | SIGXFSZ | SIGXFSZ | | SIGXFSZ |
| | | SIGEMT | | |
| **INTERRUPT** | SIGINT | SIGINT | SIGINT | SIGINT |
| | SIGTERM | SIGTERM | SIGTERM | SIGTERM |
| | SIGHUP | SIGHUP | | SIGHUP |
| **ERROR** | SIGABRT | SIGABRT | SIGABRT | SIGABRT |
| **DUMP** | SIGQUIT | SIGQUIT | | SIGQUIT |
| | | | SIGBREAK | |

If a signal is not handled by the JVM, the operating system take 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 is:

```
JAVA_DUMP_OPTS="ONcondition(dumptype[count],
dumptype[count]),ONcondition(dumptype[count],...),
USERABEND(nnnn),ceedumpoptions"
```

where dumptype can be:

- ALL
- NONE
- JAVADUMP (see Chapter 25, “Using Javadump,” on page 219)
- SYSDUMP
- CEEDUMP
- HEAPDUMP (see Chapter 26, “Using Heapdump,” on page 245)

If **USERABEND** is set, it must specify an integer 1 through 4094, or it will be ignored. If set, it takes precedence over all other settings and for any condition, the LE will terminate the JVM with the specified abend code, and bypass any cleanup

routines. You can intercept this abend code by using a SLIP TRAP to take a system dump. No other dump processing is done because no return is produced.

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 pertaining to each register. The default options for CEEDUMP are:

```
"THREAD(ALL),PAGESIZE(0),ENC(CUR),NOENTRY,GENO"
```

Additional (or overriding) options can be appended to the JAVA_DUMP_OPTS string. Parsing of these is left to right, so whatever is specified last takes precedence.

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.

**Note:** Setting **TERMTHDACT(UADUMP)** causes the JVM to explicitly disable all signal handling. This behavior has come about for historical reasons and might be changed in future.

Dumps are produced in the following form:

- SYSDUMP: On TSO as a standard MVS data set, using the default name of the form: &userid.JVM.TDUMP.&jobname.D&date.T&time, or as determined by the setting of the **JAVA_DUMP_TDUMP_PATTERN** environment variable, the syntax of which is: %s.JVM.TDUMP.&JOBNAME..D&YYMMDD..T&HHMMSS″
- CEEDUMP: In the current directory, or as determined by the setting of _CEE_DMPTARG as: CEEDUMP.&date.&time.&processid
- JAVADUMP: In the same directory as CEEDUMP, or standard JAVADUMP directory as: JAVADUMP.&date.&time.&processid.txt
- HEAPDUMP: The heapdump is written to a file that is named ″HEAPDUMP.yyyymmdd.hhmmss.txt″ in the current directory.

## AIX

Dumps are produced in the following form:

- SYSDUMP: Output is written to a core file named core.&processid. &timestamp.txt that is in the current working directory.
- JAVADUMP: Output is written a file named javacore.&processid.&timestamp.txt. See Chapter 25, “Using Javadump,” on page 219 for more information.
- HEAPDUMP: The heapdump is written to a file that is named ″heapdump<pidnumber>.<time>.txt″ in the current directory.

## Windows

In Windows, if the JVM terminates following its own dump processing, the default signal handler is reinstated and the terminating signal is passed to the operating system, which then runs whatever application debug tool is specified in the registry (for the AeDebug key). This tool would typically be, for example, the MS Developer Studio debugger or Dr.Watson.

Dumps are produced in the following form:

- SYSDUMP: Output is written to a file named core.&processid.&timestamp.dmp into the same directory that is used for JAVADUMP.
- JAVADUMP: Output is written to a file named a file named javacore.yyyymmdd.hhmmss.pidnum.txt. See Chapter 25, “Using Javadump,” on page 219 for more information.
- HEAPDUMP: The heapdump is written to a file that is named ″heapdump.yyyymmdd.hhmmss.txt″ in the current directory.

## Linux

Dumps are produced in the following form:

- SYSDUMP: Not available.
- JAVADUMP: Output is written to a file named `javacorettttttttt.pppp.txt`. See Chapter 25, “Using Javadump,” on page 219 for more information.
- HEAPDUMP: The heapdump is written to a file that is named ″heapdump<pidnumber>.<time>.txt″ in the current directory.

# Chapter 28. 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
- Method input parameters
- Method return value

However, method trace has the following restrictions:

- Not all platforms support the 'input parameter' and 'return value' trace.
- Sometimes the 'input parameters' and 'return value' traces do not work with the JIT turned on.

If you ask for input parameters but they cannot be produced because of JIT interaction, you will not see an error. Try turning the JIT off if you do not see all the output that you expect.

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 321.

This chapter describes the basic use of trace. Use this chapter to learn the basic use of trace. Once you feel comfortable using trace, see Chapter 33, "Tracing Java applications and the JVM," on page 321 for more detailed information.

## Running with method trace

Control method trace by using command-line parameters that specify system properties. To specify a system property, use the `-D<property name> =<property value>` option on the command line. You can find several examples of how to use the method trace properties below.

All the method trace properties are of the format `ibm.dg.trc.<something>`. The set of these properties is quite large and is fully described Chapter 33, "Tracing Java applications and the JVM," on page 321.

If you want method trace to be formatted, set two properties:

- **ibm.dg.trc.print** — set this property to 'mt' to invoke method trace.
- **ibm.dg.trc.methods** — set this property to decide what to trace.

The first property is only a constant: `-Dibm.dg.trc.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:

```
ibm.dg.trc.methods=[[!]method_spec[,...]]
```

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".

The parentheses, (), that are in the method_spec define whether or not to trace method parameters.

## Examples of use

- **Tracing entry and exit of all methods in a given class:**

  ```
  -Dibm.dg.trc.methods=ReaderMain.*
  -Dibm.dg.trc.methods=java/lang/String.*
  ```

- **Tracing entry, exit and input parameters of all methods in a class:**

  ```
  -Dibm.dg.trc.methods=ReaderMain.*()
  ```

- **Tracing all methods in a given package:**

  ```
  -Dibm.dg.trc.methods=com/ibm/socket/*.*()
  ```

- **Multiple method trace:**

  ```
  -Dibm.trc.dg.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**

  ```
  -Dibm.dg.trc.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, however, it can be rather slow. To work around this, you can make the method trace output appear in a compressed binary form and thus minimize its impact on performance. You can then redirect this form of the output to an output file, rather than only to the console as in the description above.

You use a tool, supplied with the IBM JVM, to analyze and dump the output binary file. You can even route trace to your own plug-in agent and process it at will (see Chapter 33, “Tracing Java applications and the JVM,” on page 321).

## Real example

```
java -Dibm.dg.trc.methods=ReaderMain.*(),ConferenceUI.*() -Dibm.dg. trc.print=mt ReaderMain
```

Results:

```
JVMDG200: Diagnostics system property ibm.dg.trc.print=mt
JVMDG200: Diagnostics system property ibm.dg.trc.methods=ReaderMain.*(),ConferenceUI.*()
09C000      0075D878   > Bytecode method ReaderMain.<clinit> This = 0290D918
09C001      0075D878   < Exiting  method ReaderMain.<clinit>
09C000      0075D878   > Bytecode method ReaderMain.main This = 0290D918 Arguments
: Array of type "java/lang/String" = 100ADC50
09C000      0075D878   > Bytecode method ReaderMain.<init> This = 0298CD38 Arguments: Integer = 0
09C001      0075D878   < Exiting  method ReaderMain.<init>
09C000      0075D878   > Bytecode method ReaderMain.traceLevel This = 0290D918
09C001      0075D878   < Exiting  method ReaderMain.traceLevel
09C000      0075D878   > Bytecode method ReaderMain.traceOn This = 0290D918
09C001      0075D878   < Exiting  method ReaderMain.traceOn
09C000      0075D878   > Bytecode method ReaderMain.loadData This = 0298CD38
09C000      0075D878   > Bytecode method ReaderMain.traceLevel This = 0290D918
09C001      0075D878   < Exiting  method ReaderMain.traceLevel
09C000      0075D878   > Bytecode method ReaderMain.traceLevel This = 0290D918
09C001      0075D878   < Exiting  method ReaderMain.traceLevel
09C000      0075D878   > Bytecode method ReaderMain.traceLevel This = 0290D918
09C001      0075D878   < Exiting  method ReaderMain.traceLevel
09C001      0075D878   < Exiting  method ReaderMain.loadData
09C000      0075D878   > Bytecode method ConferenceUI.<clinit> This = 0290D728
09C001      0075D878   < Exiting  method ConferenceUI.<clinit>
09C000      0075D878   > Bytecode method ConferenceUI.<init> This = 02911528 Arguments
: Type "EventSemaphore" = 04ECB250
09C001      0075D878   < Exiting  method ConferenceUI.<init>
09C000      0BEC3B78   > Bytecode method ConferenceUI.run This = 02911528
09C000      0BEC3B78   > Bytecode method ConferenceUI.shewWindow This = 02911528
09C000      0BEC3B78   > Bytecode method ConferenceUI.buildTree This = 02911528
09C001      0BEC3B78   < Exiting  method ConferenceUI.buildTree
09C000      0BEC3B78   > Bytecode method ReaderMain.traceLevel This = 0290D918
09C001      0BEC3B78   < Exiting  method ReaderMain.traceLevel
09C001      0BEC3B78   < Exiting  method ConferenceUI.shewWindow
09C001      0BEC3B78   < Exiting  method ConferenceUI.run
09C000      0BC51AB8   > Bytecode method ConferenceUI.windowActivated This = 02911528 Arguments
: Type "java/awt/event/WindowEvent" = 04F2F8A8
```

The first two lines show that the JVM has accepted the diagnostics properties.

Taking the third line as an example, the remaining lines comprise:

- `09C000`, an internal JVM trace point used by some advanced diagnostics.
- `0075D878`, 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 remaining fields show whether a method is being entered (>) or left (<), followed by details of the method.

# Chapter 29. Using the dump formatter

The cross-platform dump formatter is the main debugging tool for the IBM JVM. It is a Java tool that runs on all the IBM supported platforms. The dump formatter is not a debugger; it is a dump analysis tool that is usually used to analyze crash dumps.

The dump formatter has two main modes:

- Console mode (the default).
- GUI mode (Swing based using JInternalFrames): This mode is invoked through the **-g** parameter that, in this book, is called "Dumpviewer".

The dump formatter takes an SDFF file as input or, for z/OS, an SVC dump. An SDFF file is created by running **jextract** on a core dump. This SDFF file contains all the information that relates to the JVM. Because of the information about types, the dump formatter knows all about the JVM data structures. It can interpret the data that is in the dump and present it in a form that can be understood. The dump formatter makes it easier to:

- Look inside the JVM
- Look at the state of the heap
- Trace down the loaded classes and methods
- Trace through the stack of the JVM threads
- Check the locks state
- Find what addresses refer to

It is unlikely that the most powerful features of the dump formatter are of much use to anyone who does not have detailed JVM-internals knowledge, JVM source code, and build map information. These items are not available to external customers. However, used simply as a thread and memory viewer, the dump formatter is still a powerful tool.

**Note:** When a Java application fails, the production of system dumps is ***not*** enabled by default. See Chapter 12, "First steps in problem determination," on page 97 for details on how to enable the dump.

This chapter begins with summaries of:

- "What the dump formatter is" on page 262
- "Dump formatter dumps" on page 262 and
- "How to use the dump formatter" on page 262

The main part of the chapter fully describes:

- "Analyzing dumps with jformat" on page 263

The chapter ends with "Dumpviewer" on page 286 and "Analyzing dumps with Dumpviewer" on page 291.

## What the dump formatter is

You can run the dump formatter on any platform and you can use it to look at a dump that is taken on from any platform. For example, you can look at z/OS dumps on a Windows platform.

The dump formatter consists of:

**jextract**
On all the platforms, except z/OS, platform-specific extractors take a system-produced dump (see individual platform problem determination sections) and convert it into a common format that can be used by jformat and Dumpviewer. The extractor utility is normally run on the system on which the dump was produced. The jextract utility is in the jre\bin directory of your Java installation

**jformat**
This is a Java-based application that takes the common format file that is produced by the extractors (or, for z/OS, the direct dump and the typesfile) and provides several commands that allow you to analyze the dump more easily. The jformat application is in the sdk\bin directory of your SDK installation.

**Dumpviewer**
This is a Java Swing-based application that provides most of the same functions as jformat, but uses a GUI-based interface instead of a console-based one. Dumpviewer has menu items that drive panels (JInternalFrames) that correspond to the commands that are invoked in console mode. Dumpviewer is launched by jformat -g. For further information on Dumpviewer see "Dumpviewer" on page 286 and "Analyzing dumps with Dumpviewer" on page 291.

An important characteristic of the dump formatter is that it provides flexibility for extension by using plug-ins to enable the support of different types of dump files, and for the addition of new dump analysis commands.

## Dump formatter dumps

The dump formatter takes as input a dump file that is formatted into a platform-independent form. For example, a dump that is taken on a Linux platform can therefore be debugged on a Windows workstation. Each platform, apart from z/OS, has code that enables it to take a relevant dump (for example, a core dump on Linux) and translate it into the common format (known as SDFF). z/OS does not require an SDFF dump because z/OS works directly on native dumps.

## How to use the dump formatter

To use the dump formatter:

1. Obtain a relevant platform dump, as described in Part 3, "Problem determination," on page 95 for each platform. The platform dumps are usually created by the native operating system and are acceptable for analysis by native tools.
2. On platforms other than z/OS, convert the platform dump into a common format dump. The tool that translates from the platform-specific format (for example, Linux core file or Windows minidump) is called the dump extractor and is installed in the jre/bin subdirectory of your Java installation as jextract.

The syntax of jextract is:

```
jextract [options ][-o outFilename ][-f executablePathName ]coreFilename
```

where options include:

`-verbose`: Enable verbose output

`-?` or `-help`: Print this help message.

`-Xt [level ]`: Set trace level

`-Xheader`: Generate only SDFF header.

`coreFilename` is the system dump file that you want to process.

In general:

- The -Xt, -Xheader and -verbose options are unlikely to be relevant unless a service representative asks you use them.
- You need the -f option only if the executable file is not Java, or is not found in its usual place.
- You do not need the -o option if you want the output file to be named as coreFilename.sdff.

**Attention:** The SDFF file will contain:

- All the data from the core file, that data having been reformatted to a platform-independent form
- Data that has been extracted from the executable file and the libraries that are in use
- Java specific data that needs to be derived in place

The SDFF might therefore be a large file. Allow for at least the size of the input core file, and some extra.

3. Start the dump formatter and load the dump from step 2 on page 262. Steps 1 and 2 must be performed on the platform where the dump was produced. Step 3 can be performed anywhere.

The IBM Java service team almost always requires a dump formatter dump (the output from step 2 above) or a z/OS SVC dump when you submit a problem.

## Analyzing dumps with jformat

jformat is a console-based application that allows you to analyze dumps that are produced by Java applications. jformat supports two types of dump format:

- SDFF format dumps, which are produced by use of provided extractors from the normal system dumps
- MVS svc dumps (or TDUMP)

The range of dump formats that are supported might be extended in the future. Likely extensions include the addition of more dump plug-ins and the expansion of the range of commands supported and analysis options.

For z/OS, if you type ″help″ under jformat, you see a set of commands that apply to the analysis of a z/OS SVC dump. For z/OS dumps to invoke jformat with a different Java address space, use the following command:

```
java -Xbootclasspath/p:jformat.jar -Dsvcdump.default.asid=<asid>
com.ibm.jvm.dump.format.DvConsole -d <dumpname>
```

## Minimum requirements and performance considerations

The suggested minimum requirements for successfully running jformat are:

- A 500 MHz pentium or equivalent
- 256 MB RAM
- 10 GB of free hard disk space

The above requirements have been tested successfully under Windows/NT to handle and analyze dumps of up to 2 GB with reasonable performance.

Some operations are processor and I/O intensive, and can take a long time to complete. For example, the **dis os** (display object summary) command scans the various heaps that are present in the dump and constructs a names index file on disk. (Note that the second time you use **dis os**, it runs much faster because the existing index file is reused and not recreated.) Depending on the size of dump and the number of items that are on the heaps, this operation can take a long time to complete. Using the above configuration, for example, the scanning of a 650 MB MVS dump that contains 800,000 objects takes between 15 and 30 minutes. Obviously, for exceedingly large dumps that contain large numbers of objects, a more powerful machine is better. You might want to start the console by using the Java **-Xmx** parameter to ensure that conditions do not occur in which you do not have enough memory:

```
java -Xmx<size> com.ibm.jvm.dump.format.DumpFormat
```

(However, in testing, the default memory that is taken by Java when it is running jformat has been found to be enough.)

## Installing jformat

The IBM SDK for Java v1.4.2 supplies (and supports) the dump formatter. You do not have to install it explicitly.

## Starting jformat

Usage:

```
jformat [[-d]dumpfilename] [other options]
```

where options include:

`-d dumpfilename`: The dump to format

`-g`: Bring up Dumpviewer (GUI)

`-o outputfilename`: Output to file

`-i inputfilename`: Input from file

`-t tracefilename`: Trace to file

`-T`: Trace to console

`-w`: Set working directory

`-?` or `-h`: Help

Typing `jformat -h` to get information about the usage of jformat.

## Opening the dump

If you launch jformat without specifying a dump on the command line, jformat is initialized with a default set of commands that is followed by a ″ready″ prompt.

When a dump is identified (either on the command line or if you use the set dump= command) jformat loads additional command plug-ins. The resulting output is similar to the following:

```
jformat
*** Loading plugins ***
loaded com.ibm.jvm.dump.plugins.DvBaseCommands

*** All plugins loaded successfully ***

Ready......


set dump=e:\windump.sdff
 ......command executing

Extracting windump.sdff.hdr (1 tick = 327680 bytes) .../
Dump recognised as a new format SDFF dump
Confirmed that permissions allow names index file to be written.
Dump successfully opened

Suffix established as sov
   Will now attempt load of Dvsov.properties
   to find supplemental command plugins

Dvsov.properties found

*** Loading plugins ***
loaded com.ibm.jvm.dump.plugins.DvBaseCommands
loaded com.ibm.jvm.dump.plugins.DvGeneralSov
loaded com.ibm.jvm.dump.plugins.DvMonitorsSov
.............. lines removed for clarity .....
loaded com.ibm.jvm.dump.plugins.DvStorageCommands
loaded com.ibm.jvm.dump.plugins.DvHeapDumpPlugin

*** All plugins loaded successfully ***


Using typedefs from core...


Sanity check passed (use "SANCHK Verbose" for details)

STANDALONE JVM:
  address:             0x1015da40
  currently_in_GC0:    false
  signal received:     21

fullVersion: J2RE 1.4.2 IBM Windows 32 build cndev-20040408

Ready......
```

**Note:** within the dump there is embedded information that allows the dump formatter to understand the internal control block structures in the JVM. This information is referred to as *typedefs information* or *typesfile*.

## Command plug-ins

jformat is extensible. New commands that enable new function can be added. You do this by using command plug-ins that extend the CommandPlugin class.

Available command plug-ins are controlled by using the CommandPlugins stanza within the Dv.properties file. Existing plug-ins are constantly being enhanced to support new commands.

Function is included that provides a sanity check when a dump is opened (the JVM eyecatchers are checked), and handles multiple JVM sets in an address space. Further sanity checks will be added in the future.

Available plug-ins are:

- DvBaseCommands
- DvBaseFmtCommands
- DvTraceFmtPlugin
- DvObjectsCommands
- DvClassCommands
- DvJavacorePlugin
- DvXeCommands
- DvHeapDumpPlugin

The commands that are supported by each plug-in are described below.

## Shortened command forms

When possible, jformat tries to recognize commands in their shortened forms. For example, jformat recognizes a full command such as `display memory 0x12345678` if you specify it as `dis mem 0x12345678` or even as `d m 0x12345678`. To see details of the full command formats that are recognized by each plug-in, use the command `dis pl verbose`. The tables below show some valid short forms.

*Table 12. Shortened command forms for jformat*

| Verb | Short Verb |
|---|---|
| DISPLAY | DIS |
| HELP | ? |
| FORMAT | FOR |
| FINDNEXT | FN |
| FINDPTR | FP |
| WHATIS | W |

*Table 13. Shortened modifier forms for jformat*

| Verb Modifier | Short Form | |
|---|---|---|
| SYSTEM | SYS | (used by DIS) |
| MEMORY | MEM | (used by DIS) |
| THREAD | T | (used by DIS) |
| INT | I | (used by DIS) |
| LONG | L | (used by DIS) |
| POINTER | P | (used by DIS) |
| PROCESS | PROC | (used by DIS) |
| EXECENV | EE | (used by FORMAT) |
| STGLOBAL | STG | (used by FORMAT) |
| CLASSSUMMARY | CLS | (used by DIS) |
| CLASS | CL | (used by DIS) |
| METHODS | ME | (used by DIS) |

*Table 13. Shortened modifier forms for jformat (continued)*

| Verb Modifier | Short Form | |
|---|---|---|
| OBJECT | OBJ | (used by DIS) |
| LOCKSUMMARY | LS | (used by DIS) |
| LOCKEDOBJECTS | LO | (used by DIS) |
| OBJECTSUMMARY | OS | (used by DIS) |
| JAVASTACK | JS | (used by DIS) |
| NATIVESTACK | NS | (used by DIS) |
| JITMETHODS | JITM | (used by DIS) |

## Supported commands

The tables below lists the commands that jformat supports. Note that they are not case-sensitive. Optional parameters are placed in square brackets [].

### Commands from DvBaseCommands

*Table 14. Commands from DvBaseCommands for jformat*

| Verb | Modifier | Parameter format | Note | Example |
|---|---|---|---|---|
| SET | | | 1 | Set |
| SET | <key name> | | 2 | Set <key name> |
| SET | <key name> | = | 3 | Set <key name>= |
| SET | <key name> | = <value> | 4 | Set <key name> = <value> |
| SET | DUMP | = <dump id> | 5 | Set dump=p:\sdff32 |
| SET | ERROR | = file | 24 | set error=my.txt |
| SET | FORMATFILE | = file | 22 | |
| SET | FORMATAS | = a (ASCII) or e (EBCDIC) | | |
| SET | OUT | = file | 23 | set out=my.txt |
| SET | OUTPUT | | | |
| SET | WORKDIR | = x:\yyy | 27 | |
| SET | TRACE | = [ON]/[OFF] | 28 | |
| SET | ASID | | 30 | |
| SET | JVM | | 31 | |
| SET | PROC | | 32 | |
| SET | THREAD | | 33 | |
| | | | | |
| DISPLAY | MMAP | | 6 | Dis mmap |
| DISPLAY | MEMORY | ([@],<address>,[<length>]) | 7 | Dis memory(804c000,256) |
| DISPLAY | THREAD | [(<thread id>)] | 9 | Dis thr(1234) |
| DISPLAY | INTEGER | ([@], <address>) | 12 | Dis int(0x804c000) |
| DISPLAY | LONG | ([@], <address>) | 13 | Dis long(@804c000) |
| DISPLAY | POINTER | ([@], <address>) | 14 | Dis ptr(804c000) |
| DISPLAY | PROCESS | [(<process id>)] | 15 | Display proc(666) |
| DISPLAY | SYSTEM | | 17 | Dis sys |

*Table 14. Commands from DvBaseCommands for jformat (continued)*

| Verb | Modifier | Parameter format | Note | Example |
|---|---|---|---|---|
| DISPLAY | as | | 29 | |
| DISPLAY | plugin | [verbose] | | Dis pl [verbose] |
| | | | | |
| FIND | *See note 18* | target[,sa][,ea][,bdry][,ps][,limit] | 18 | Find java,804c000,,,,10 |
| FINDPTR | *See note 19* | | 19 | FindPtr 804c000 |
| FINDNEXT | *See note 20* | | 20 | FindNext |
| | | | | |
| + | *See note 8* | | 8 | |
| - | *See note 8* | | 8 | |
| | | | | |
| DISPLAY | HINTS | | 21 | Display HINTS |
| WHATIS | | | 26 | WHATIS OAB 3 4 1 |
| HELP | | | 25 | Help (or ?) |

**Notes:**

1. The **SET** command by itself displays the set values for all keys.
2. **SET <key name>** displays the value for the specified key.
3. **SET <key name>=** sets the value of the specified key to null.
4. **SET <key name>=<value>** sets the value of the specified key to the contents of `<value>`. You can also use substitution values in `<value>` by using the $ character. For example:

   ```
   set a=999
   set b=abc
   set c=$a$b
   ```

   leaves c with a value of 999abc. Substitution applies to all commands. Therefore, `dis mem $c` would transform into `dis mem 999abc` if you used the example above.
5. **Set dump=**. This command determines which dump is being worked on. It takes an argument of a file location and discovers which of the dump plug-ins handles this dump. For SDFF format dumps, additional (unseen) processing determines the current address space, process, and thread. It also determines the location of the JVM and STGlobalPtr (used for heap analysis).
6. **Display mmap** displays the memory ranges for the current address space.
7. **Display memory([@] <address>, [<length>])** supports '@'. Therefore, `display memory(@123456,500)` would find a pointer at address 123456 and use this pointer as the starting point of where to display 500 bytes of memory. Parameter `<address>` is required and is in hex format. The parameter `<length>` is optional and defaults to 256.
8. When you have displayed a section of memory, use the '+' and '-' keys to display the next or previous pieces of memory.
9. **Display thread** displays the current thread. The current thread is set when the dump is opened (**SET DUMP=**), but you can change it by using the **SET THREAD** command.
10. **Display thread(<thread name>)** displays information for the specified thread.

11. **Display thread(*)** displays information for all threads that are associated with the current process.
12. **Display integer([@], <address>)** displays the value of the Java integer (32 bits) at the given address. It takes account of the endian-ness of the machine and can be shortened to **dis int**. Therefore, by using **dis int** in a little-endian system, where memory is displayed, `0x00000010` would be 16777216, as an integer, not 16.
13. **Display long([@], <address>)** displays the value of the Java long (64 bits) at the given address. As with **Display integer**, this command takes account of the endian-ness of the machine.
14. **Display pointer([@], <address>)** displays the pointer representation from the address given; this takes account of whether the system is 32- or 64-bit and whether it is big endian or little endian.
15. **Display process** displays information about the current process. This includes a list of all the threads that are in this process and any environment data or data on loaded modules that is included in the dump.
16. **Display process (<process id>)** displays information on the specified process.
17. **Display system** displays information about the provided dump; whether the system is 32- or 64-bit and the whether it is big endian or little endian, the number of address spaces and processes, and the type of system and subsystem that are being used.
18. The **Find <target>** command and its associated commands **FindPtr** and **FindNext** are used to find items in memory. The target identifies what to find. It can be either a string, such as java, or a hex pattern, such as 0x12345. (Note that hex patterns are always padded to an even number of characters with 0's.) The second and third parts of the command are optional. They are respectively a start address (such as 804c000) and an ending address to delimit the memory searched. The fourth part of the command is optional also (defaults to 1); it is the byte boundary on which the given target should start. The fifth and sixth parts of the command (both optional) specify respectively the number of bytes of the first found target to display (default 256) and the maximum number of matches to find (maximum 32,767). Missing parts are indicated by a comma with no following value. The default range of memory that is scanned for the target is all the memory that is available.
19. The **FindPtr** command has the same syntax that **Find** has, except that the target is a hex address. Therefore, `FindPtr 804c000` searches for the pointer 804c000 in memory.
20. The **FindNext** command repeats the previous **Find** but with the start address just beyond the one that was used by the previous **Find**.
21. **Display HINTS** displays the HINTS that are established as part of the **Set dump=** command. For more information, see "Hints" on page 276.
22. **Set formatfile** tells jformat to use a specified format file (also known as a types file) to interpret successfully the contents of the JVM that is in the dump. you can use this to override the file that is normally established when a dump is opened.
23. **Set out** directs the command output towards a file. **set out=*** redirects the output back to the screen.
24. **Set error** directs error and diagnostic output toward a file. **set error=***redirects the output back to the screen.
25. **Help** shows general help for the commands that are available for each plug-in.

26. **Whatis** queries the various command plug-ins for what they know about an object (normally an address).
27. **Set workdir** allows dumps on read-only file systems to be used. It stores the nidx files (produced when dis os is run) on the working directory.
28. **Set trace** starts or stops debug trace from inside jformat.
29. **Display as** displays a list of address spaces, processes, and threads.
30. **Set asid** allows more functions on systems that contain multiple address spaces (mainly z/OS)
31. **Set jvm** allows switching between multiple JVMs in the same address space.
32. **Set proc** allows the process in the address space to be set (assume multiple processes per address space).
33. **Set thread** allows the current thread (for an address space or process) to be established.

## Commands from DvBaseFmtCommands

*Table 15. Commands from DvBaseFmtCommands for jformat*

| Verb | Modifier | Parameter format | Note | Example |
|---|---|---|---|---|
| FORMAT | execenv | | | Format execenv |
| FORMAT | jvm | | | Format jvm |
| FORMAT | <address> | Number | 1 | Format 804c000 as Jvm |
| | | | | |
| DISPLAY | HINTS | | 2 | DISPLAY HINTS |
| DISPLAY | CB | | 3 | |
| DISPLAY | CBO(x) | | 4 | |

**Notes:**

1. The formatting of an address as a control block does not perform consistency checking to ensure that a block of memory matches its believed usage. The control block name that is being used is case-sensitive.
2. **Display Hints** displays the saved memory addresses that can be used in the **Format** command. Therefore:

   ```
   Format xyz as def
   ```

   would first look for a hint that is called ″xyz″, obtain its value, and use that as the address to format.
3. **Display CB** displays the control blocks that are available for formatting memory.
4. **Display CBO(x)** displays a control block and the offsets of the fields.

## Commands from DvTraceFmtPlugin

The Trace Format Plug-in extracts trace records from the in-memory trace buffers that are contained in the dump file, creates a trace file (.trc) from this, then formats it.

All trace plug-in commands are invoked by Trace.

*Table 16. Commands from DvTraceFmtPlugin for jformat*

| Verb | Modifier | Parameter format | Note | Example |
|---|---|---|---|---|
| TRACE | FORMAT | | 1 | trace format |

*Table 16. Commands from DvTraceFmtPlugin for jformat (continued)*

| Verb | Modifier | Parameter format | Note | Example |
|---|---|---|---|---|
| TRACE | EXTRACT | | 2 | trace extract |
| TRACE | summary | | 3 | trace summary |
| TRACE | help | | 4 | trace help |
| TRACE | set | none | 5 | trace set |
| TRACE | set | option=value | 6 | trace set indent=true |
| TRACE | indent | true, false or none | 7 | trace indent false |
| TRACE | verbose | true, false or none | 7 | trace verbose true |
| TRACE | symbolic | true, false or none | 7 | trace symbolic |
| TRACE | threads | See Notes | 7, 8 | Trace threads 0x2545bc8 |
| TRACE | entries | See Notes | 7, 9 | Trace entries awt,LK |
| TRACE | datadir | See Notes | 7, 10 | Trace datadir=c:\temp |
| TRACE | display | | 11 | Trace display - |
| TRACE | display | + | 11 | Trace display + |
| TRACE | display | - | 11 | Trace display - |
| TRACE | display | page=40 | 11 | Trace display page=40 |

**Notes:**

1. If the `Trace Extract` command has not yet been run, it runs automatically and the extracted.trc file is used as input to the formatter.
2. This command creates a trace file, called extracted.trc, in the current directory. This file contains the contents of JVM in-memory trace buffers and the required header as if the trace had been taken on a running JVM.
3. Run `Trace extract` or `Trace format` before you run this command.
4. This command produces a page of help about trace commands.
5. This command lists all the environment variables that are used to control the formatting of trace.
6. Each of the environment variables that can be set by using their setter command below (that is Threads, indent, entries) can also be set with the syntax `trace set option=value`.
7. With no parameters, these commands display current setting of the option. With a parameter, the option is set to that value.
8. With no parameters, this command displays the threads that are being traced. The parameters can be one or more comma-separated hex thread ids. (The valid ids can be listed by using `Trace threads`.)
9. This command selects a subset of the tracepoints from the trace file that will be formatted. For a list of the components that can be selected, enter `TRACE ENTRIES ?`
10. This command is used to set the location of the TraceFormat.dat file (tracepoint definitions). This is usually only of interest if you are formatting a dump that is from one release of Java, but you are using a different release to run the dump formatter. Datadir specifies the directory where TraceFormat.dat is. If the file is not there, the command is rejected.
11. This is a primitive browser that displays the first or current page (25 lines) of the formatted trace. You must first have formatted the trace by using the `trace`

`format` command. `Trace display +` displays the next page and `Trace display -` displays the previous page. `Trace display page=nn` sets the pagesize to nn lines.

## Commands from DvClassCommands

The DvClassCommands Format Plug-in displays details of the classes that the JVM had loaded. The details from these classes are used by the DvObjectCommands plug-in to enable formatting of object instances.

*Table 17. Commands from DvClassCommands for jformat*

| Verb | Modifier | Parameter format | Note | Example |
|---|---|---|---|---|
| DIS | cls | [filters] | 1 | Dis cls(Test2Frame) |
| Dis | cl | filter1[,filter2][,etc....] | 2 | Dis cl(!*a*,b*) |

**Notes:**

1. The **DVClass** command causes the loadedClasses, loadedACSClasses, and loadedSystemClasses portions of the JVM to be scanned, and the addresses of all the classes to be stored in memory. In addition, for each class a short summary record is output. If no parameter is specified, every loaded class is displayed. Using a filter causes only those classes that are handled by the filter to be displayed; therefore, **dis cls(!*a*)** displays only those classes that do not contain an "a" in their name.
2. If the **dis cls** command has not been run, the **DVClass** command runs automatically (and nonverbosely) before it actions the **dis cl** command that was input. **Dis cl(*)** displays data on all the classes that are in the dump (and produces more output than **dis cls** does). You can use filters to control the output. Additionally, **dis cl** allows enhancer keywords to follow the filter. The allowed keywords are **me** (for methods), **st** (for statics), and **fld** (for fields). By using these enhancers, you give more information for the class. Therefore, **dis cl(java/lang/String) fld** expands the details for the java/lang/String class to show the nonstatic field names and types and their offsets in the instance record for the object. "Example session" on page 276 shows an example of the output from this.

## Commands from DvObjectsCommands

The DvObjectsCommands Plug-in displays details of the objects and locks that are present in the dump.

*Table 18. Commands from DvObjectsCommands for jformat*

| Verb | Modifier | Parameter format | Note | Example |
|---|---|---|---|---|
| DIS | os | | 1 | Dis os |
| DIS | obj | filter1[,filter2][,etc....] | 2 | Dis obj(!*a*,b*) |
| DIS | obj | 0xhhhhhhhh | 2 | Dis obj 0x4afe000 |
| DIS | ls | | 3 | Dis ls |
| DIS | lo | filter(s) | 4 | Dis lo(java/awt/EventQueue) |
| DIS | loc | filter(s) | 5 | Dis loc(*$Lock) |
| DIS | lr | | 6 | |
| DIS | lt | | 7 | |
| | | | | |
| DEADLOCK | | | 8 | |

**Notes:**

1. On first use (which might be an invocation that was caused by the use of another command), the **Dis os** command causes a scan of all the objects that are in the heaps, and produces a summary that is based on this information. The initial scan can take a long time, but is necessary so that objects can be located by later commands. "Example session" on page 276 shows the output from this command.
2. **Dis obj** displays details for one or more objects. The parameters take two forms: either normal filter parameter structure, or an object address specification (0xhhhhhh). Details of the object and its instance data are displayed in the example session.
3. **Dis ls** displays a summary of the lock information, including monitor pool information, registered monitors, and thread identifiers. See "Commands from DvJavaCore" for more info.
4. **Dis lo** displays more detailed information about a lock, and formats some relevant control blocks.
5. **Dis loc** shows locking status for all objects or (by using a filter) a selection of objects.
6. **Dis lr** displays registered monitors.
7. **Dis lt** displays locks by thread ID.
8. **Deadlock** performs deadlock analysis and identifies blocking threads or objects.

## Commands from DvJavaCore

*Table 19. Commands from DvJavaCore for jformat*

| Verb | Modifier | Parameter format | Note | Example |
|---|---|---|---|---|
| JAVACORE | | | 1 | JAVACORE |
| JAVACORE | HELP | | 2 | JAVACORE HELP |
| JAVACORE | TAGS | true \| false | 3 | JAVACORE TAGS TRUE |
| JAVACORE | SECTION | | 4 | JAVACORE SECTION LK |
| JAVACORE | VERBOSE | true \| false | 5 | JAVACORE VERBOSE FALSE |

Note that JAVADUMP can be used instead of JAVACORE.

**Notes:**

1. **JAVACORE** with no parameters produces a full Javacore.
2. **JAVACORE HELP** displays valid commands.
3. **JAVACORE** tags are switched on or off. The default is true ( = on).
4. Valid section names are TITLE, XHPI, CI, DC, DG, ST, XE, LK, XM, CL, END.
5. **VERBOSE** allows more information to be obtained about command progress.

## Commands from DvXeCommands

*Table 20. Commands from DvXeCommands for jformat.*

| Verb | Modifier | Parameter format | Note | Example |
|---|---|---|---|---|
| DISPLAY | JAVASTACK | | | DIS JS |
| DISPLAY | NATIVESTACK | | | DIS NS |
| DISPLAY | JITMETHODS | | | DIS JITM |

See "Example session" on page 276 for an example of the output of these commands.

## Commands from DvHeapDumpPlugin

You can now get a Heapdump (see Chapter 26, "Using Heapdump," on page 245) from a system dump. A Heapdump is a list of objects that are in the Java heap. The advantage of this method is that you can use heap analysis tools in any condition in which a system dump occurs.

You can send this data to a compressed (GZipped) text file or across the network. Programs such as HeapRoots (see "Using the HeapRoots post-processor to process Heapdumps" on page 249) can analyze these files.

*Table 21. Commands from DvHeapDumpPlugins for jformat.*

| Verb | Modifier | Parameter format | Note | Example |
|---|---|---|---|---|
| HD | F | | Send Heapdump to a file. The filename is derived from the dump name. | HD F |
| HD | N | | Send Heapdump to the network socket. | HD N |
| HD | P | PORT | Set HD_PORT, which is the port for network send. The default port is 21179. | HD P 2001 |
| HD | H | HOST | Set HD_HOST, which is the host for network send. The default host is "localhost". | HD H myhost |

**Note:** You must run the **dis os** command to make available these commands from DvHeapDumpPlugins.

**Example output:**

```
Sample output:

Ready......
hd f

 ......command executing
Tue Sep 02 09:57:30 GMT 2003: Opening "SR19.TDUMP.txt.gz" to write heapdump ...
        Tue Sep 02 09:57:32 GMT 2003: All 8737 objects done. Closing file ...
Heap Dump finished.

Ready......
```

**If you want to read the compressed file manually,** or use it with a program that does not support GZip compressed input files, run:

```
gunzip -c original | head     # to look at start of file using 'head'
gunzip -c original > newfile  # to uncompress file and create 'newfile'
```

## Control block formatting

To format control blocks in memory, you must know the typedefs that are associated with the specific level of the JVM. The JVM has been changed to incorporate this information into memory. Therefore, when you identify a dump (`SET DUMP= <command>`) the typedefs are located and you should see the following message:

```
Using typedefs from core
```

If the typedefs cannot be found in memory, the **FORMATFILE** setting is used to provide the information that is required to format control blocks. For example, `SET FORMATFILE=my.file` locates my.file and uses it to provide the necessary information about control block layout and structure.

**Note:** If the control block information does not match the structures that are in memory, it is likely that the commands to analyze the dump will fail.

## Settings

Various keyword values are set during jformat initialization and on the opening of a dump. Several of these values establish the context for various commands. This section describes the most important of those values.

**ASID, PROCESS**, and **THREAD** are all initialized when you invoke **Set dump=**. If multiple address spaces are present in the dump, `Set asid=xxx`, where `xxx` is the id of the address space, resets the PROCESS and THREAD values to values that are relevant to that address space.

**DUMP**. The command **Set dump=xxx** opens the dump source and establishes the interface (that is, the dump plug-in) between the dump source and jformat. Until you invoke a **Set dump**, other commands will not produce much useful detail. (Therefore, if you do not have a dump, no memories are available to look at.)

**FORMATAS**. By default, the **dis mem** command shows memory that is interpreted on ASCII or EBCDIC, depending on the one that is suitable for the system that generated the dump. To overrule this:

- For ASCII, use `SET FORMATAS=A`
- For EBCDIC, use `SET FORMATAS=E`.

Similarly, the **FIND** command assures that the string to be found is in the form that is suitable for the system. You can overrule this also by using **FORMATAS**.

**CURRJVM**. On most systems, only one JVM exists in a dump. On systems such as z/OS CICS, however, several JVMs can exist in a dump. When a dump is opened, jformat can detect whether multiple JVMs exist. If they do, jformat sets the value of **CURRJVM** to the value of the first JVM that it finds. You can use the **set JVM=** command to change the value of **CURRJVM**. For example, if multiple JVMs exist, you have values for JVM#1, JVM#2, JVM#3, and so on. To select which JVM is to be analyzed, give the command:

```
set jvm=$jvm#n
```

where `n` is a value 1, 2, 3, and so on.

## Dump plug-ins

jformat uses dump plug-ins to support multiple dump formats.

Two formats are supported. For AIX, Linux, and Windows, the SDFF format is supported. This is the format that is produced when the extractors are run.

Additionally, for z/OS, the direct IPCS dump is supported.

The supported dump plug-ins are identified by the DumpPlugin section in the DV.properties file.

## Property files

Several property files are used to control jformat.

Dv.properties is a generic repository for customizing several aspects of the functioning of jformat. Its main uses are to identify the dump plug-ins and command plug-ins that are available (the `DumpPlugins=` and `CommandPlugins=` lines respectively). The properties file DvSetDefaults.properties controls the initial variables that jformat defines. You can edit this file. For example, you can:

- Establish shortcuts for commands you use regularly
- Set your display width

## Hints

Hints are values that are set when a dump is opened and when memory is being explored. Hints enable jformat to access common fields easily and to remember their location when found. When a dump is first opened, the execenv for the active task is examined and used to access the JVM (by means of jvmP field), which is then used to evaluate various other important fields that are in the dump. The Display Hints command shows what hints are in place at any time. Typically after accessing a dump, the output should look like this:

```
STGLOBALPTR = 403229c0
ALLOCBITS = 40571000
CURHEAPMAX = 103ffbfc
JVM = 40321b80
LOCKINTERFACE = 40321b94
CURHEAPMIN = 100001fc
```

## Example session

An example of the use of jformat is shown below. Note that the output has been edited for compactness. Lines that are in italics are those that were entered on the command line.

```
*=========================================================*
* This is an annotated jformat session with comments
* enclosed in boxes like this -user input is bold italicised
* and command output is in plain text.Some of the output
* is shortened for clarity and replaced with an indicator
* that lines have been cut out.
*
* jformat is an evolving entity so don 't be suprised
* if the output differs slightly in future upgrades
*
* It all starts with the user typing jformat,which can
* be found in sdk \bin ,,,,,,,,,,,,,,,,,,,,,,
*=========================================================*
jformat

*** Loading plugins ***
loaded com.ibm.jvm.dump.plugins.DvBaseCommands

*** All plugins loaded successfully ***
```

```
Ready......

*=========================================================
* The "set dump="command identifies the dump that we
* will use.This command queries the available dump
* plugins to see which one handles this types of dump
* In this case its in sdff format. This command also
* loads other command plugins relevent to this dump
*=========================================================
set dump=e:\windump.sdff
 ......command executing

Extracting windump.sdff.hdr (1 tick = 327680 bytes) .../
Dump recognised as a new format SDFF dump
Confirmed that permissions allow names index file to be written.
Dump successfully opened

Suffix established as sov
   Will now attempt load of Dvsov.properties
   to find supplemental command plugins

Dvsov.properties found

*** Loading plugins ***
loaded com.ibm.jvm.dump.plugins.DvBaseCommands
loaded com.ibm.jvm.dump.plugins.DvGeneralSov
loaded com.ibm.jvm.dump.plugins.DvMonitorsSov
.............. lines removed for clarity .....
loaded com.ibm.jvm.dump.plugins.DvStorageCommands
loaded com.ibm.jvm.dump.plugins.DvHeapDumpPlugin

*** All plugins loaded successfully ***

*=========================================================*
* The processing invoked by "set dump="will normally
* establish the typedefs to use for formatting control
* blocks from that held in memory and there output "Using
* typedefs from core".If the typedefs are not present
* in the dump then you can use "set FORMATFILE="to
* identify a typedefs file that corresponds to this
*  dump's level.
*=========================================================*

Using typedefs from core...

*=========================================================*
* The dump you have loaded is given a quick check (sanity
* check)to make sure that it is not obviously corrupt.
*=========================================================*
Sanity check passed (use "SANCHK Verbose" for details)

STANDALONE JVM:
  address:             0x1015da40
  currently_in_GC0:    false
  signal received:     21

fullVersion: J2RE 1.4.2 IBM Windows 32 build cndev-20040408

Ready......


*=========================================================*
*
* "dis system"and "dis proc"show assorted information
* about the general nature of the dump being handled
*=========================================================*
dis system
```

```
 ......command executing

   System Summary
   ==============

fullVersion: J2RE 1.4.2 IBM Windows 32 build cndev-20040408
  32 bit - Little Endian
  Number of Address Spaces: 1
  Number of Processes     : 1
  System                  : Windows
  SubSystem               : Windows 2000
  Processor (number)      : ?(?)
  Processor subtype       : ?
  Current process id      : ?
  Number of JVMs found    : 1

Ready......

dis proc

 ......command executing

Process Information
===================
Architecture: 32 bit - Little Endian


AddressSpace: 0   Process: 0
Signal      : 00000000............

 Thread:      0x394  ExecEnv: 0x00235170  Thread name: main
  Thread: 7d8  Not a java thread
 Thread:      0x730  ExecEnv: 0x0b1e30d8  Thread name: Signal dispatcher
 Thread:      0x45c  ExecEnv: 0x0b1f2650  Thread name: Reference Handler
 Thread:      0x330  ExecEnv: 0x0034d588  Thread name: Finalizer
  Thread: 3b4  Not a java thread
 Thread:      0x708  ExecEnv: 0x0b2beed0  Thread name: DG event write thread

Environment Variables
=====================

ALLUSERSPROFILE=C:\Documents and Settings\All Users
APPDATA=C:\Documents and Settings\philr\Application Data
CLASSPATH=.;e:\sdk\bin
.............. lines removed for clarity .....
windir=C:\WINNT
_NT_SYMBOL_PATH=SRV*D:\MS_Symbols*http://msdl.microsoft.com/download/symbols
IBM_JAVA_COMMAND_LINE=jformat

Loaded data
==============
        E:\sdk\bin\jformat.exe
                 at 0x400000    length=36864(0x9000)
        C:\WINNT\system32\ntdll.dll
                 at 0x77f80000  length=503808(0x7b000)
.............. lines removed for clarity .....
                 at 0xb110000   length=73728(0x12000)
        E:\sdk\jre\bin\jitc.dll
                 at 0xb3f0000   length=3809280(0x3a2000)
        E:\sdk\jre\bin\DBGHELP.DLL
                 at 0xbf40000   length=733184(0xb3000)
        C:\WINNT\system32\PSAPI.DLL
                 at 0x690a0000  length=45056(0xb000)


Ready......
```

```
*=========================================================*
* The "dis cls"command can be used to display the current
* set of loaded classes or can be used with filters (as
* shown)to limit the information shown.Without the
* filter summary information for all 360 classes would
* have been output.
*=========================================================*
dis cls
 ......command executing

 Classes Summary
 ===============

 Number of classes found via loadedClasses       = 0
 Number of classes found via loadedACSClasses    = 0
 Number of classes found via loadedSystemClasses = 405

Class:com/ibm/jvm/dump/format/DvConsole$DvCommand
  @0xb27eb38  version:48.0
  super:java/lang/Object  @0x290278
  methods:@0xbaf8498   method count:1
  Class Loader Initialized: true
  fields:@0xbaf83c0   fields count:9   instance size:36
  statics:@0xbaf8510   statics count:0

Class:java/util/Observable
  @0xb27ea08  version:48.0
  super:java/lang/Object  @0x290278
  methods:@0xbc2e7b0   method count:10
  Class Loader Initialized: true
  fields:@0xbc2e780   fields count:2   instance size:8
  statics:@0xbc2ec38   statics count:0

.............. lines removed for clarity .....

Class:java/lang/Class
  @0x290148  version:48.0
  super:java/lang/Object  @0x290278
  methods:@0xb134020   method count:86
  Class Loader Initialized: true
  fields:@0xb133e70   fields count:18   instance size:40
  statics:@0xb136820   statics count:9

Class:java/lang/NoClassDefFoundError
  @0x290018  version:48.0
  super:java/lang/LinkageError  @0x293a48
  methods:@0x34ba30   method count:2
  Class Loader Initialized: true
  fields:@0x0   fields count:0   instance size:16
  statics:@0x34bb18   statics count:0

Ready......

*=========================================================*
* "Dis os"(which is often run under the covers by other
* commands)traverses the various heaps and records the
* position of all object instances.The dump used here
* is a small one and hence scanning is relatively fast,
* but for larger dumps with larger numbers of objects
* then this command can take a large amount of time!!
* Since this is the first time this dump was used then
* the "nidx"file has been created this is used to allow
* fast access to individual objects on subsequent uses
* of jformat thus avoiding the need to completely
* retraverse the heaps
*=========================================================*
dis os
```

```
 ......command executing

Invalid or non-existant class names index file
Dump must be scanned (dis os) and index created

Traversing Thread Local Heaps

        MH-TLH cache block located at 0x2a6b698 for thread 394 ee=0x235170


        4 of 5 java threads did not have TLH cache blocks
        5 of 5 java threads did not have TH-TLH cache blocks
TLH finished - 195 objects


Traversing the Middleware heap


  12:05:56 5000 objects processed...(10% of range scanned so far)

  12:06:05 10000 objects processed...(14% of range scanned so far)

  12:06:13 15000 objects processed...(17% of range scanned so far)

Mid traversal finished - 17317 objects


Traversing the transient heap

Event traverseHeap min value not in dump
Traversing the system heaps


Sys 0 traversal finished - 199 objects


Sys 1 traversal finished - 215 objects



Analysis of Monitors started....


Analysis of Monitors ended.



 Objects Summary
 ===============

MH-main
        start address: 0x2a6b698
        end   address: 0x2a7a69c
        heap  size   : 0xf004  61444
        Objects found : 195

Mid
        start address: 0x28c01fc
        end   address: 0x30bfbfc
        heap  size   : 0x7ffa00  8387072
        Objects found : 17317

Sys 0
        start address: 0xb270014
        end   address: 0xb27ec64
        heap  size   : 0xec50  60496
        Objects found : 199
```

```
Sys 1
        start address: 0x290014
        end   address: 0x29ff64
        heap  size   : 0xff50  65360
        Objects found : 215

ACS Heap
        start address: 0x0
        end   address: un-measured (non meaningful)
        Objects found : 0

 Total number of objects found via all Heaps     = 17926

 *** The JVM does not seem to be in Garbage Collection (not within GC0).



 Total number of "Swapped" objects         = 0 (0 bytes)
 Total number of "Locked" objects          = 4
 Total number of Hashed and Moved objects  = 0
 Total number of Hashed objects            = 0
 Total number of Arrays                    = 8458


0000000001 (=0000000016 bytes) of : arrObj java/security/Principal
0000000001 (=0000000016 bytes) of : arrObj java/security/cert/Certificate
0000000001 (=0000000016 bytes) of : arrObj java/text/FieldPosition
0000000001 (=0000000016 bytes) of : com/ibm/jvm/io/LocalizedInputStream$1
0000000001 (=0000000016 bytes) of : com/ibm/misc/BASE64Decoder
.............. lines removed for clarity .....
0000000262 (=0000953456 bytes) of : array byte
0000000414 (=0000125856 bytes) of : java/lang/Class
0000000478 (=0000009792 bytes) of : arrObj java/lang/Class
0000000504 (=0000012096 bytes) of : java/lang/StringBuffer
0000000550 (=0000017600 bytes) of : java/util/HashMap$Entry
0000000641 (=0000025280 bytes) of : arrObj java/lang/Object
0000000747 (=0000023904 bytes) of : java/util/Hashtable$Entry
0000001086 (=0000037608 bytes) of : arrObj java/lang/String
0000005560 (=0000177920 bytes) of : java/lang/String
0000005675 (=0000409264 bytes) of : array char

 Total number of objects = 17926

 Total byte count = 1912032
Finished..

Ready......

*=========================================================*
* Having scanned the heaps it 's now possible to show the
* details of individual object instances using the dis obj
* command.In the example shown there is only one instance
* of DvObjectsCommands in the heap -so dis obj 0x910460
* would have given the same result.Note that super
* classes get expanded.
*=========================================================*
dis obj(com/ibm/jvm/dump/plugins/DvBaseCommands)
 ......command executing

==========================================

 @ 0x292b6d0 (com/ibm/jvm/dump/plugins/DvBaseCommands)   (heap: Mid)

    ===== Super Class expansion for: com/ibm/jvm/dump/plugins/CommandPlugin
(76) method      instance of Ljava/lang/reflect/Method; @ 0x0
(80) paramString         instance of Ljava/lang/String; @ 0x0
```

```
                 Null String <<>>
(84) verb        instance of Ljava/lang/String; @ 0x0
                 Null String <<>>
(88) verbModifier        instance of Ljava/lang/String; @ 0x0
                 Null String <<>>
(92) verbModifierForFind         instance of Ljava/lang/String; @ 0x0
                 Null String <<>>
(96) seperator   instance of Ljava/lang/String; @ 0x0
                 Null String <<>>
(100) enhancers  instance of Ljava/util/Vector; @ 0x0
(104) forcedEnd  boolean: false (0x0)
(108) addReady   boolean: false (0x0)
(112) cpr        instance of Lcom/ibm/jvm/dump/plugins/CommandPluginResponse; @ 0x2a69360
(116) cpInProgress       boolean: true (0x1)
(120) hasOnDumpIdentified        boolean: false (0x0)
    ===== Super Class expansion for: java/lang/Thread
(0) name         Array of char @ 0x2a6b468
        <Thread 0>
(4) priority     integer: 5 (0x5)
(8) threadQ      instance of Ljava/lang/Thread; @ 0x0
(16) eetop       long: 0 (0x0)
(24) single_step         boolean: false (0x0)
(28) daemon      boolean: false (0x0)
(32) userDaemon  boolean: false (0x0)
(36) started     boolean: false (0x0)
(40) target      instance of Ljava/lang/Runnable; @ 0x0
(44) group       instance of Ljava/lang/ThreadGroup; @ 0x2938390
(48) contextClassLoader  instance of Ljava/lang/ClassLoader; @ 0x292b7b0
(52) inheritedAccessControlContext       instance of Ljava/security/AccessControlContext; @ 0x2a6b43
0
(56) threadLocals        instance of Ljava/lang/ThreadLocal$ThreadLocalMap; @ 0x0
(60) inheritableThreadLocals     instance of Ljava/lang/ThreadLocal$ThreadLocalMap; @ 0x0
(64) stackSize   long: 0 (0x0)
(72) blocker     instance of Lsun/nio/ch/Interruptible; @ 0x0

Ready......

*=========================================================*
* "Dis cb" shows what control block names  can be used
*  with the "format addr as controlblock" command.
*  "dis cbo(name) allows the structure of a particular
*  control block to be  investigated.
*=========================================================*
dis cb
 ......command executing


AFrameInterface
ArrayOfChar
ArrayOfObject
AssertionList
BTEntry
.............. lines removed for clarity .....
utTraceCfg
utTraceControl
utTraceFileHdr
utTraceListener
utTraceRecord
utTraceSection
utf8_bucket_t

Ready......

dis cbo(utf8_bucket_t)
 ......command executing

utf8_bucket_t
```

```
----------
Offset   Name   Type
------   ----   ----
0(0)     next   ....
4(4)     hash   ....
8(8)     length ....
c(c)     status ....
10(10)   from_cb....
14(14)   utf8   ....


Ready......

*=========================================================*
* You can of course display memory as shown below.There
* are various variations on the theme to show integers
* longs and pointers taking due account of endianess and
* whether the system was 32 or 64 bit.
* The example below also shows use of "set formatas" to
* force the display of ascii or ebcdic.
*=========================================================*
dis mem 10000,64
 ......command executing

00010000: 3D003A00 3A003D00 3A003A00 5C000000 | =.:.:.=.:.:.\...
00010010: 3D004300 3A003D00 43003A00 5C004400 | =.C.:.=.C.:.\.D.
00010020: 6F006300 75006D00 65006E00 74007300 | o.c.u.m.e.n.t.s.
00010030: 20006100 6E006400 20005300 65007400 |  .a.n.d. .S.e.t.

Ready......
set formatas=e

 ......command executing

Ready......

dis mem 10000,64
 ......command executing

00010000: 3D003A00 3A003D00 3A003A00 5C000000 | ............*...
00010010: 3D004300 3A003D00 43003A00 5C004400 | ............*...
00010020: 6F006300 75006D00 65006E00 74007300 | ?....._...>.....
00010030: 20006100 6E006400 20005300 65007400 | ../.>...........

Ready......

*=========================================================*
* There are also commands for looking at locks,such as
* dis ls (lock summary)shown below....
*=========================================================*
dis ls
......command executing

LOCKING INFORMATION:

Inflated Object-Monitors:
     Information is from a table of inflated monitors:
         monitor_index_cb_t 0x1015c8d0

(0x235760)
    (0x2940598) java/lang/ref/Reference$Lock
        <unowned>
            Waiting to be notified:
                   0x45c  "Reference Handler"

(0x2357b0)
    (0x29402d0) java/lang/ref/ReferenceQueue$Lock
```

```
        <unowned>
            Waiting to be notified:
                   0x330  "Finalizer"


Registered Monitors:
 Pointer to first registry monitor (0x1015af64)

(0xb2ba398) JITC PIC Lock

(0xb2b1318) JITC CHA lock

 .............. lines removed for clarity .....

(0x236c18) Namespace Cache subpool lock

(0x236bd8) Class Storage subpool lock

(0x236b98) CL Tables subpool lock

(0x27afe8) JIT General subpool lock



Thread Identifiers:

0x394  "main"

0x730  "Signal dispatcher"

0x45c  "Reference Handler"

0x330  "Finalizer"

0x708  "DG event write thread"



Flat & Inflated Object-Monitors:

    (0x29402d0) java/lang/ref/ReferenceQueue$Lock
        <unowned>
            Waiting to be notified:
                   0x330  "Finalizer"

    (0x2940598) java/lang/ref/Reference$Lock
        <unowned>
            Waiting to be notified:
                   0x45c  "Reference Handler"

    (0x29450e8) java/io/BufferedInputStream
        flat locked by 0x394 "main", entry count 0

    (0x2a16c78) java/io/InputStreamReader
        flat locked by 0x394 "main", entry count 1

No deadlocks detected

Finished..

Ready......

*=========================================================*
* ....and commands for looking at threads.....
*     and their stack details
*=========================================================*
dis t
```

```
 ......command executing

    ASID: 0 PROCESS: 0 THREAD: 394


Info for thread - 394
=============================================
  Name   : main
  Id     : 394
      Use "dis ns" and "dis js" to display stacks
  ExecEnv: 235170
  jvmP   : 0x1015da40

   cs      0000001b   ds      00000023   eax     00290608   ebp     0006f380
   ebx    00000000   ecx     0000001d   edi     00170688   edx     00000000
   eip     77f8917a   es       00000023   esi     0006f3b0   esp     0006f364
   flags   00000246   fs      00000038   gs      000000000   ss      00000023

    ASID: 0 PROCESS: 0 THREAD: 394

Ready......

dis js
 ......command executing

 Java stack for thread 394 - main
 ==========================

        at java.io.FileInputStream.readBytes(Native method)
        at java.io.FileInputStream.read(FileInputStream.java:219)
        at java.io.BufferedInputStream.read1(BufferedInputStream.java:236)
        at java.io.BufferedInputStream.read(BufferedInputStream.java:293)
        at java.io.FilterInputStream.read(FilterInputStream.java:107)
        at sun.nio.cs.StreamDecoder$ConverterSD.implRead(StreamDecoder.java:324)
        at sun.nio.cs.StreamDecoder.read(StreamDecoder.java:222)
        at java.io.InputStreamReader.read(InputStreamReader.java:207)
        at java.io.BufferedReader.fill(BufferedReader.java:152)
        at java.io.BufferedReader.readLine(BufferedReader.java:315)
        at java.io.BufferedReader.readLine(BufferedReader.java:378)
        at com.ibm.jvm.dump.format.DvConsole.<init>(DvConsole.java:371)
        at com.ibm.jvm.dump.format.DvConsole.main(DvConsole.java:822)

Ready......

dis ns
 ......command executing

 Native stack for thread 394 - main
 ============================

        at 0x77f8917a in ZwRequestWaitReplyPort+0xb (C:\WINNT\system32\ntdll.dll)
        at 0x77f891d2 in CsrClientCallServer+0x55 (C:\WINNT\system32\ntdll.dll)
        at 0x7c5aa3e7 in OpenConsoleW+0x244 (C:\WINNT\system32\KERNEL32.DLL)
        at 0x7c5aa4dc in ReadConsoleA+0x2b (C:\WINNT\system32\KERNEL32.DLL)
        at 0x7c586134 in ReadFile+0x80 (C:\WINNT\system32\KERNEL32.DLL)
        at 0x7801bd9c in putch+0x9c (C:\WINNT\system32\MSVCRT.dll)
        at 0x7801bf5c in read+0x72 (C:\WINNT\system32\MSVCRT.dll)
        at 0x3a6725 in _sysRead+0x15 (E:\sdk\jre\bin\hpi.dll)
        at 0xb544aef in _reserve_m_block+0x1df
(E:\sdk\jre\bin\jitc.dll)
        at 0xb544c7d in _jit_mem_alloc+0xad (E:\sdk\jre\bin\jitc.dll)
        at 0xb54562f in _jit_code_mem_alloc+0x1f (E:\sdk\jre\bin\jitc.dll)
        at 0xb40c251 in _register_committed_code+0x31 (E:\sdk\jre\bin\jitc.dll)
        at 0x10042c25 in _classLoaderLink+0x185 (E:\sdk\jre\bin\classic\jvm.dll)

Ready......
```

```
*=========================================================*
* The "whatis"command can be used to establish whether
* something is an address and if it is whether it points
* to anything interesting...as below ...this feature
* will be enhanced in future releases....
*=========================================================*

w 0x292b6dc
 ......command executing
 Its an address
 Address "0x292b6dc" is present in this dump
  "0x292b6dc" is in heap "Mid"
 found object (com/ibm/jvm/dump/plugins/DvBaseCommands) at 292b6d0 (length:136)
        ....that covers this address (offset 0xc)

Ready......

*=========================================================*
* And finally you use the q (quit)command to get out!
*
* The above session has missed out on many aspects
* of jformat - there are a lots more fcommands available.
*=========================================================*
q

 ......command executing

E:\sdk\bin>
```

## Dumpviewer

Dumpviewer is the graphical version of the jformat application. It uses Swing. To start Dumpviewer, use the -g option on the **jformat** command:

```
jformat -g
```

This command starts Dumpviewer and creates this display:

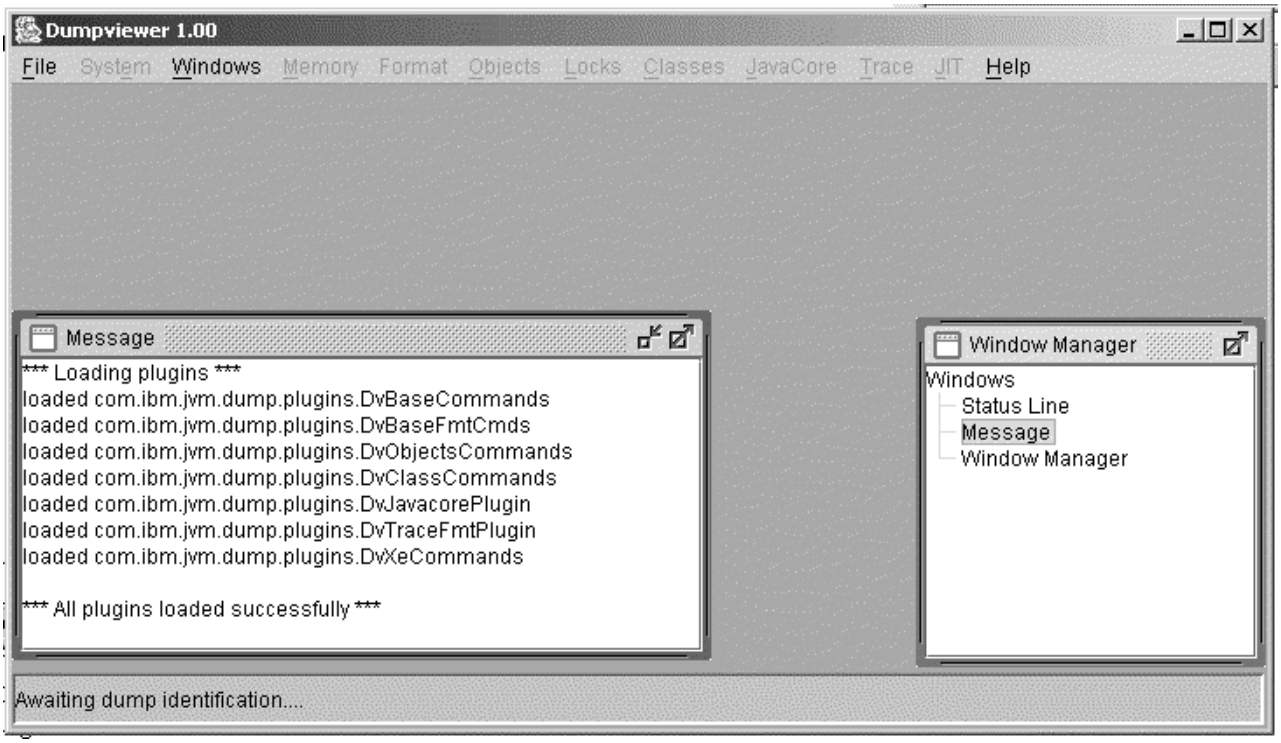



*Figure 8. First Dumpviewer display*

**Notes:**

1. If Javahelp™ is not installed on your machine, this display is preceded by a dialog box that tells you that the display is for information only and that help is not available for this session of Dumpviewer. You can download the Javahelp package from http://java.sun.com/products/javahelp.
2. To run Dumpviewer under the Unix (instead of Windows), you need a graphics-enabled environment (XWindows).

The first Dumpviewer display (see Figure 8) shows the general layout. At the top of the display is a set of pulldown menus that, when activated, display a list of selectable menu items. When Dumpviewer is in its initial state, many of these menus are inactive (grayed out). When you identify a dump by using the **Open** menu item or by choosing a file from the history list at the bottom of the pulldown (see Figure 9 on page 288), those menus become active.

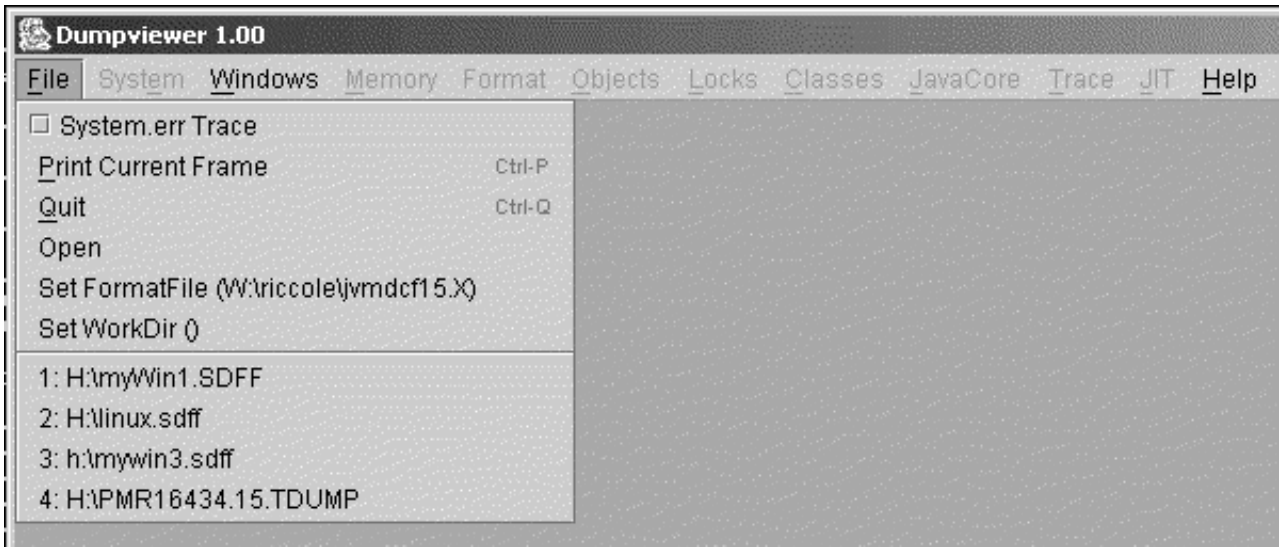



*Figure 9. Menu items and history list*

In addition to the menu bar that is at the at the top of the display, Dumpviewer always has a status line at the bottom of the display and a (movable) Window Manager internal frame. When menu items are selected and used, additional internal frames become active. Figure 10 on page 289 shows what the display might look like when a dump file has been opened.

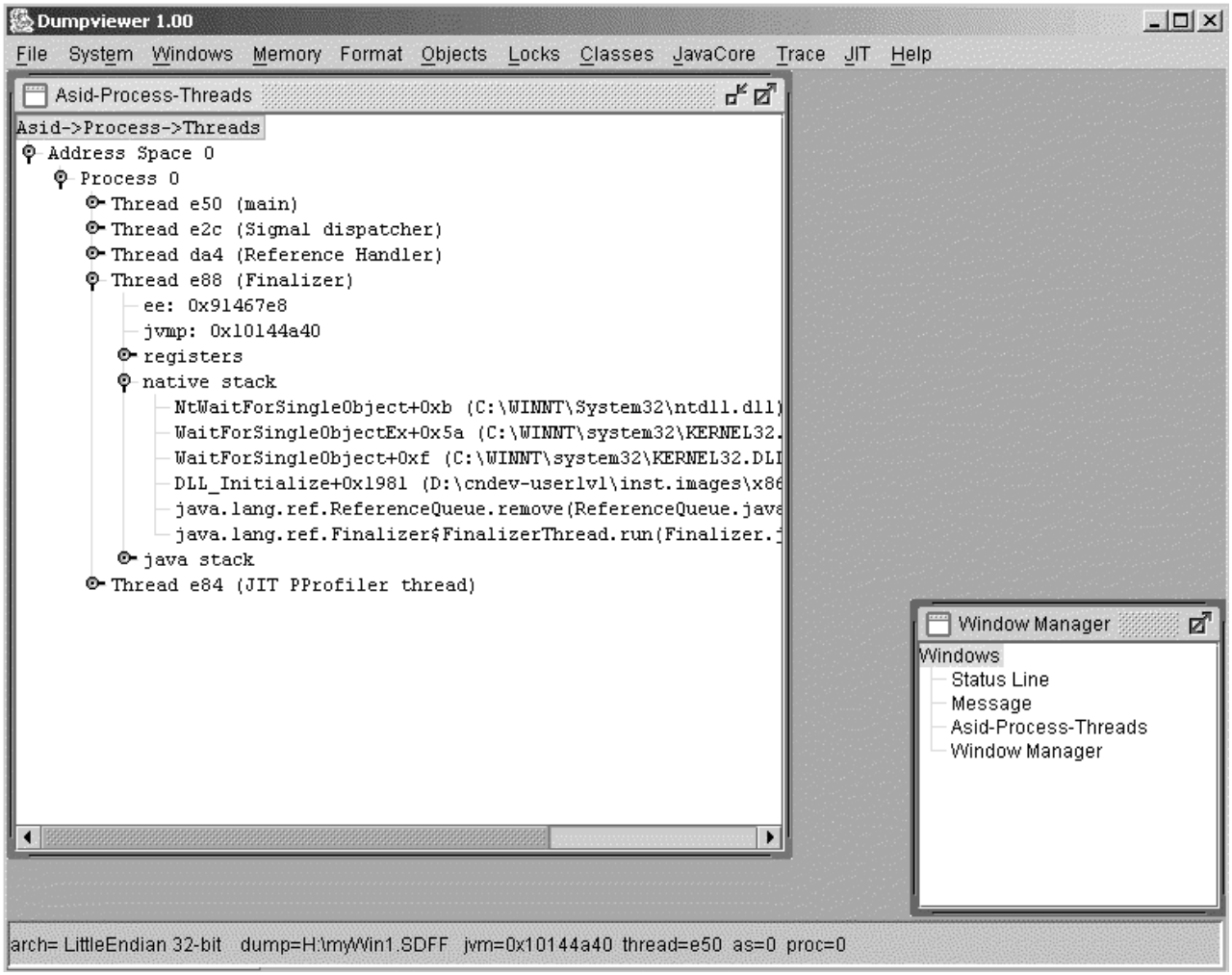



*Figure 10. The display after a dump file has been opened*

In Figure 10, the status line shows the name of the dump and information about the architecture of the dump. It shows also the currently-active address space, process, thread, and JVM that are in that dump. You can have only one dump open at a time in any one instance of Dumpviewer. Also in the display is the Asid-Process-Threads frame that is created when the dump is opened. In the example shown, the initial tree structure that is presented has been navigated and the native stack information for thread e88 is being examined.

**Note:** Many menu items cannot execute until the various object storage heaps that are in the JVM have been scanned. When such a condition occurs, a dialog box (see Figure 11 on page 290) is opened. After the scan of the heaps is completed, you must use the dialog box to reinvoke the menu item. How long the scan of the heaps takes to run depends on how many objects are stored in the heaps. With many objects, the scan can take a very long time to run.

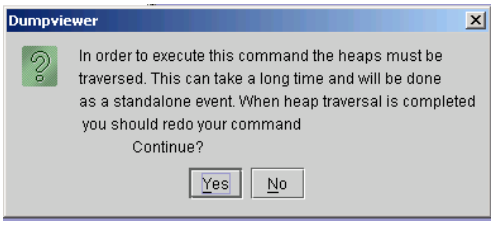



*Figure 11. Dialog box*

Dumpviewer has many items under the menubar. This book does not describe all those items. The menu bar supports menu items that allow equivalent functions for all the commands that are available in the console version. When you use more and more menu items, the screen can become very busy unless you close some frames (see Figure 12). You can use double left mouse clicks on the Window Manager panel to navigate. Many frames are available, some text based, some containing trees, some containing tables, and some with navigation controls of their own.

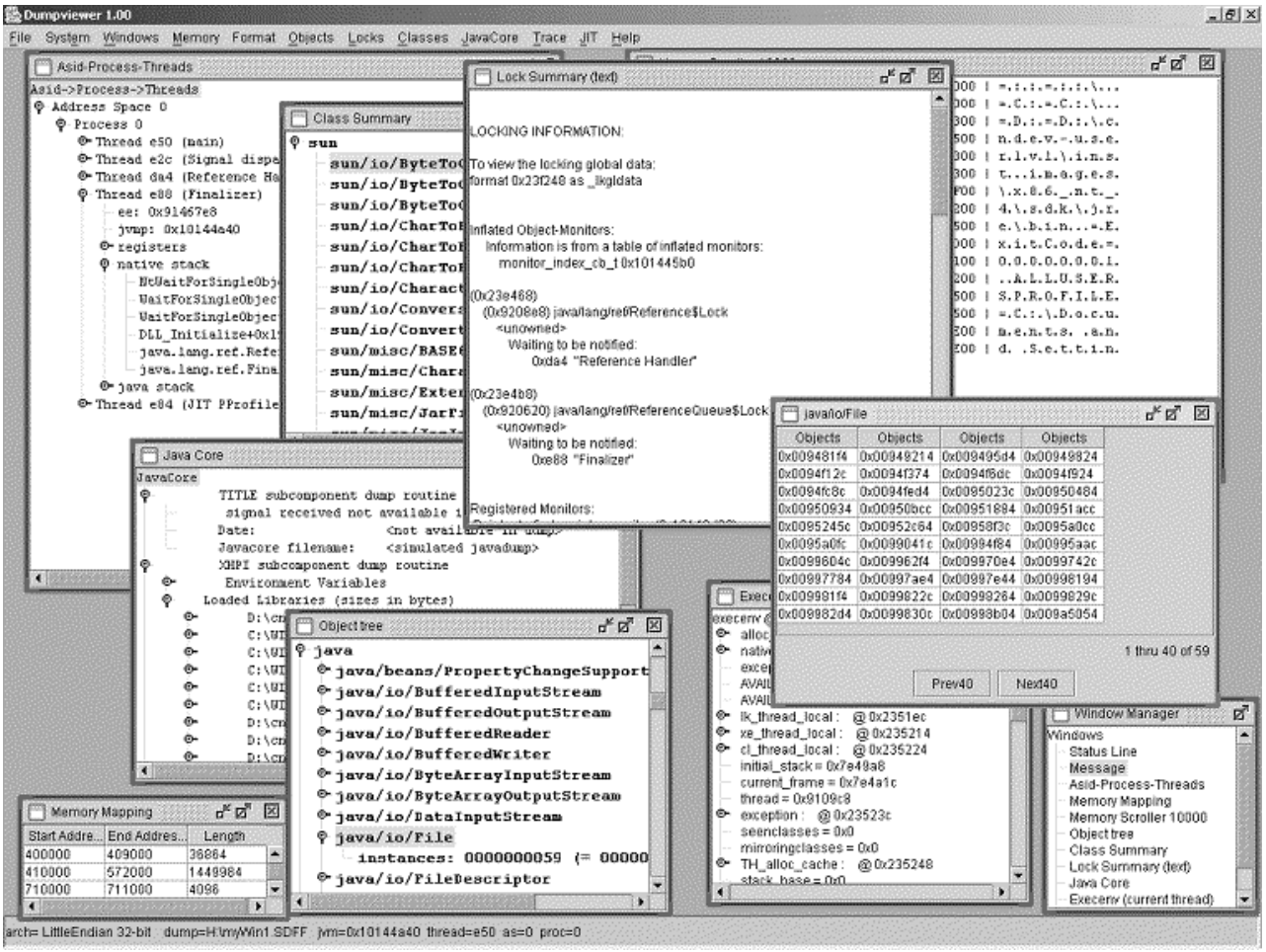
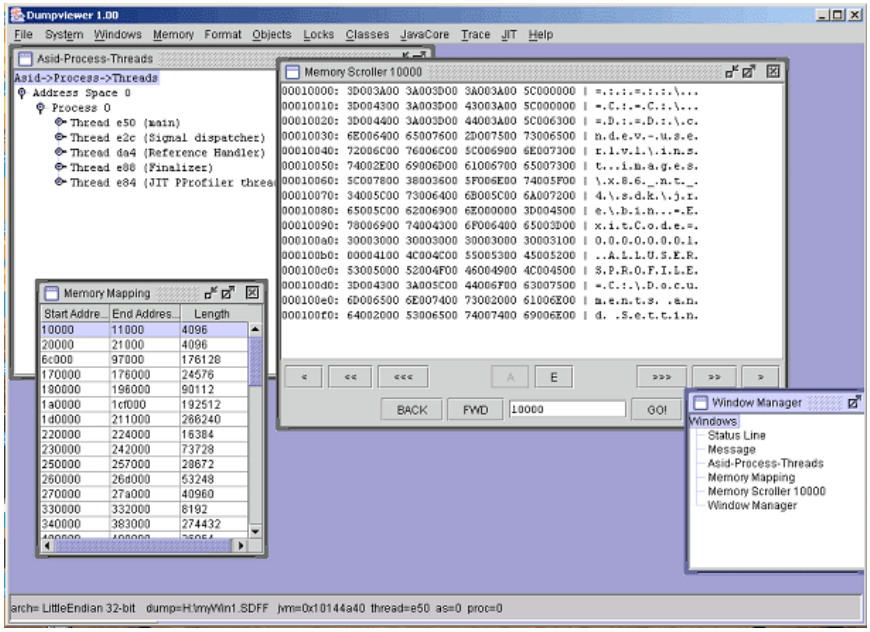



*Figure 12. A busy screen*

Note that you can run multiple instances of jformat, both graphical (-g) and console based, against the same dump.

## Analyzing dumps with Dumpviewer




Dumpviewer is the GUI-based version of jformat, which is pure Java that uses Swing constructs and makes extensive use of JInternalFrames. You invoked Dumpviewer by using the **-g** parameter on the **jformat** command. Various menu items are related to the commands that are available in the console version of jformat. Table 22 shows the relationship between the console and GUI mechanisms.

**Note:** Dumpviewer is still being developed. Although new functions are being added all the time, the levels of those functions often lag behind the levels of the equivalent functions that are in the console-based version (jformat).

*Table 22. GUI menu items and console commands for jformat*

| Menu | Item | Matching console command | Notes |
|---|---|---|---|
| File | System.err Trace | SET TRACE=ON | Trace goes to System.err |
| | Print Current Frame | -- None -- | Put the current frame out to the printer |
| | Quit | -- None -- | Ends dumpviewer |
| | Open | SET DUMP= | Choose a dump with which to work. Gives the Asid-Process-Threads view which allows display of registers, native stacks, and Java stacks. |
| | Set FormatFile | SET FORMATFILE= | What to use if no imbedded ctypes file |
| | Set WorkDir | SET WORKDIR= | Where to put nidx files |

*Table 22. GUI menu items and console commands for jformat (continued)*

| Menu | Item | Matching console command | Notes |
|---|---|---|---|
| System | Set JVM | SET JVM= | Switch to another JVM in this address space. |
| | Set Thread | SET THREAD= | Set the current thread. |
| | Set Process | SET PROC= | Set the current process |
| | Set Address Space | SET AS= | Set the current address space. |
| | Dump info | DIS SYS | |
| | Sanity Check | SANCHK | Display the sanity check verbose output. |
| Windows | Window Manager | -- None -- | Double clicking on a window jumps to it. |
| | Iconify all | --None -- | Tidy desktop. |
| | DeIconify all | -- None -- | |
| | Close all | -- None -- | Close closeable windows. |
| Memory | Map | DIS MMAP | Display a table of memory ranges and sizes. Double click to launch memory display at that position |
| | Display | DIS MEM | Scrollable display of memory |
| | Find | FIND | Find something in memory. |
| Format | Format..as.. | FORMAT xxs as... | Format an address; gives a tree. |
| | Execenv | FOR ee | Format the current threads execenv. |
| | Jvm | FOR jvm | Format the current JVM. |
| Objects | Object Summary | Dis OS | Scan the heaps. |
| | Object tree | Dis OS | Tree view of objects in heaps. Right mouse allows instances for a particular class to be displayed. |
| | Heap Analysis | -- None -- | --- future ---- |
| Locks | Lock Summary | Dis ls | Lock summary in text. |
| | Locked Objects | Dis lo | Locked objects as tree. |

*Table 22. GUI menu items and console commands for jformat (continued)*

| Menu | Item | Matching console command | Notes |
|---|---|---|---|
| | Locked Threads | Dis lt | Locked objects (by thread id) |
| | Registered Monitors | Dis lr | List registered monitors. |
| | Deadlock detection | DEADLOCK | Detect deadlocks. |
| Classes | Class Summary tree | DIS cls | Double click displays class details/methods/ fields. |
| Javacore | Javacore (tree) | Javacore | Builds simulated Javacore and displays as a tree. |
| Trace | Extract | TRACE EXTRACT | Extracts the trace from the dump. |
| | Format | TRACE FORMAT | Formats the extracted trace. |
| | Summary | TRACE SUMMARY | Gives a trace summary. |
| | Display Trace | TRACE DISPLAY | Displays the trace as a scrollable window. |
| | Spawn trace browser | -- None -- | Spawns Notepad (windows) or vi (unix) against the formatted trace |
| JIT | JIT'd methods | Dis JITM | Displays JIT'd methods from in the trace. |
| Help | Help | -- None -- | Gives access to structured help (uses JavaHelp). |
| | About | -- None -- | |

The Dumpviewer display consists of the following main areas:

- Menu bar (at the top of the screen)
- Window Manager window (at the bottom right-hand corner of screen)
- Status bar (at the bottom of the screen)
- Working area in whatever part of the screen is not covered by the previous three items)

When each menu item is selected (or in some cases, when items in existing windows are selected or double clicked), a new JInternalFrame is launched in the working area. This frame contains the information that is relevant to your task in a relevant format (straight text, JTree, JTable). Until a dump is identified, many of the menu choices are unavailable (grayed out); they become available only when the dump is identified. Other menu items (such as displaying the object tree) are not available until the heaps have been scanned. (This operation can take a long time.)

The contents of the menu bar and menu items are detailed in Table 22 on page 291. The Window Manager window is permanent and cannot be closed. It allows control to be maintained when numerous other windows (JInternalFrames) are launched. When a dump has been opened, the status bar displays the architecture of the currently-open dump, the dump name, current JVM, current thread, current address space, and current process.

**Note:** This chapter does not describe all the menu items that you can use in Dumpviewer. Neither does it describe the resulting frames of those menu items. It is intended that the help that is available in Dumpviewer will be updated to provide a sample session that better describes the functions that are available.

# Chapter 30. JIT diagnostics

A basic diagnostic test is to determine whether or not the problem is in the Just-In-Time (JIT) compiler or elsewhere in the JVM.

The JIT is tightly bound to the JVM, but is not part of it. The JIT converts Java bytecodes, which execute slowly, into native code, which executes quickly.

Occasionally, valid bytecodes might compile into invalid native code. By determining whether the JIT is faulty and, if it is, 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, and suggests some possible quick workarounds and advance bugging techniques that you can use if your problem is JIT related:



## Disabling the JIT

First, you must disable the whole JIT compiler. To disable the JIT compiler:

1. Note the current setting of the environment variable **JAVA_COMPILER**.
2. Set the variable to NONE.
3. Run the program again.
4. When the program has run, change **JAVA_COMPILER** back to its original setting.

After you have disabled the JIT, either of two conditions can exist:

- The problem remains. It is, therefore, not in the JIT. Go no further in this chapter.
- The problem disappears. It is most probably, although not definitely, in the JIT.

## Introducing the MMI

When you have disabled the whole JIT, as described in "Disabling the JIT," your program is running in purely-interpreted mode. However, when the JIT is switched on, your program is not necessarily running in purely-compiled mode. This is because the IBM JVM runs in mixed-mode interpretation (MMI) by default. The MMI is used because a large amount of Java code must be executed to start up the JVM. If all this code is compiled immediately, it takes a very long time for the JVM to start up because of the overhead in compiling all the initial methods.

The MMI maintains a threshold count for Java methods. Every time a method is called, the threshold count for that method is decremented. Until the count reaches zero, the method code is not compiled. The effect of this is to spread the compilation of methods throughout the life of an application. Some infrequently-used methods might never be compiled at all.

With the MMI switched on, therefore, at any given moment, your application consists of a mix of bytecode methods and compiled methods.

You can change the MMI threshold or even switch off the MMI completely. If the MMI is switched off, the JIT compiles every method the first time that it is called.

Therefore. the JVM can run in three JIT/MMI modes:

- **JIT on, MMI on**: The default setting. JVM starts up reasonably quickly. Runtime performance improves over the lifetime of the JVM.
- **JIT off, MMI off**: The MMI is automatically switched off if the JIT is disabled. Java runs in interpretive mode only. The JVM starts up quickly, but runtime performance is poor.
- **JIT on, MMI off**: Java runs in compiled mode only. The JVM starts up slowly, but runtime performance is satisfactory.

## Disabling the MMI

It is useful to know whether the behavior of the JIT changes depending on whether the JVM is running in mixed mode.

To disable the MMI:

1. Set the MMI threshold to zero. You set the threshold through the **IBM_MIXED_MODE_THRESHOLD** environment variable.
2. Record the setting of the **IBM_MIXED_MODE_THRESHOLD** variable, and set it to zero.
3. Ensure that the JIT is on.
4. Run your programme and check whether the fault remains consistent, changes its characteristic, or disappears completely.
5. Record the results.

## Selecting the MMI threshold

You can change the behavior of the MMI by adjusting the threshold value. If the problem disappeared when you disabled the JIT or the MMI, or if the problem characteristics changed, try changing the MMI threshold. Recommended threshold values are: 8, 20, 50, 100, and 200.

Record the behavior for each value.

## Working with MMI

If you set the environment variable **IBM_MIXED_MODE_THRESHOLD** to a value n, each method will be interpreted n times. Subsequent invocations cause the method to be compiled. However, if the method contains a loop, each iteration of the loop is considered to be an invocation of the method.

## Selectively disabling the JIT

If you think that you have a problem with the JIT, you can try more diagnostics.

The JIT provides a comprehensive set of conditional code points that you can switch in and out by using environment variables. These variables are called the JIT compile options. All these options have a name of the form `JITC_COMPILEOPT=<value>`.

By setting a value, you disable a specific part of the JIT. For example, `JITC_COMPILEOPT=NALL` disables all the functions of the JIT and causes the JIT to generate native code without doing any of the optimizations listed in "How the JIT optimizes code" on page 38.

The purpose of these options is first of all to determine whether a problem is really a JIT problem, and then to drill down by successively reducing the amount of function that is disabled. You can instruct the JIT to ignore a specific method, class, or package. Wild cards allow you to instruct the JIT not to compile methods that observe a particular pattern. For example, you can instruct the JIT not to compile any method in any class that has the name ″getItAllOn″.

The JIT compile options 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 a few lines of code that cause 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.

To summarize, the JIT compile options give you a relatively easy-to-use way to diagnose a fault and to obtain a workaround.

The basic JIT compile options are:

1. JITC_COMPILEOPT=NMMI2JIT
2. JITC_COMPILEOPT=NINLINING
3. JITC_COMPILEOPT=NQOPTIMIZE
4. JITC_COMPILEOPT=NDOPT
5. JITC_COMPILEOPT=NBCOPT
6. JITC_COMPILEOPT=NALL (a superset of all the above options)

Try each of these in turn. For example, to try the second option, set `JITC_COMPILEOPT=NINLINING`. If you want to try combining all three of the first three options, set `JITC_COMPILEOPT=NINLINING;NMMI2JIT;NQOPTIMIZE`, using the semicolon (;) as the separator for Windows. Instead of the semicolon, use the colon (:) for UNIX platforms.

Record your observations. If one of these settings causes your problem to disappear, you have a quick workaround that you can use while the Java service team are analyzing 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 parameters" on page 489), especially for those applications in which execution time is not concentrated into a small number of methods.

Also try adjusting the MMI to a value (using trial and error) for short-running applications to improve performance. Refer to "Selecting the MMI threshold" on page 296.

## Identifying JIT compilation failures

If the JVM crashes, and you can see that the failure has occurred in the JIT library (libjitc), the JIT might have failed during an attempt to compile a method. For example, you might see a line like this in the Javadump file:

```
1HPSIGRECV     SIGSEGV received in ?? at 0xa27894f5 in /usr/IBMJava2-131/jre/bin/libjitc.so.
        Processing terminated.
```

A good way to see what the JIT is doing is to use the COMPILING option of the JIT compiler; that is, JITC_COMPILEOPT=COMPILING. This option tells you when the JIT starts to compile a method, and when it ends. If the JIT seems to fail on a particular method (that is, it starts compiling, but never ends), use the other JIT compiler options to exclude the method, class, or package from the compilation (see "Advanced JIT diagnostics"). If one of these other options prevents the crash, you have an excellent workaround that you can use while the service team correct your problem.

## Advanced JIT diagnostics

Many more JIT compile options are available that, if used, give you a very good workaround and indicate the location of the problem to the Java service team. The JIT team maintain a comprehensive website that gives detailed instructions on all aspects of JIT debugging, including the more advanced compiler options.

Your IBM service representative can obtain a zip file of HTML of the JIT team's website. Unzip the files and point your browser at the top level HTML page (index.html). Follow the guide that is given there.

**Note:** The zip file is provided as-is. IBM does not provide updates automatically. It might be wise to request regular updates of this service.

# Chapter 31. Garbage Collector diagnostics

This chapter describes how to diagnose the garbage collection operation. The topics that are discussed in this chapter are:



## How does the Garbage Collector work?

You are strongly advised to read Chapter 2, "Understanding the Garbage Collector," on page 7 to get a full understanding of the Garbage Collector. A very short introduction to the Garbage Collector is given here.

The IBM JVM components include a storage manager component named "st". The st component is a memory manager, but because 99% of the complexity of memory management in the JVM is concerned with garbage collection, the term "garbage collection" is largely synonymous with memory management.

The storage component basically hands out chunks of heap space on demand. When storage runs out, (that is, the st component cannot satisfy a memory request), a memory fault occurs and garbage collection is started.

Garbage collection identifies and frees up previously-allocated chunks of heap space that are no longer being used. When this operation has been done, the storage component returns to the memory allocation request that it should now be able to satisfy.

Note that garbage collecting does ***not*** occur naturally unless, and until, a memory allocation fault occurs. The term "naturally"' refers to normal JVM operation. An application can start Garbage collection at any time, but this action is not recommended. See "How to coexist with the Garbage Collector" on page 23.

## Common causes of perceived leaks

The Garbage Collector has a record of objects that are allocated. However, the JVM cannot be notified when an application has finished with an object. Therefore, when a garbage collection cycle starts, the Garbage Collector must find in the heap all objects that are in use. When this has been done, any objects that are in the allocated records, but ***not*** in the list of in-use 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. Such a condition is normal. 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) 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 ()), and no memory allocation failure occurs. This is because Garbage Collector is a stop-the-world heavy operation. The Garbage Collector is designed to run as infrequently as possible and for as a short time as possible.

### Objects with finalizers

Objects that have finalizers cannot be collected until the finalizer has run. The Garbage Collector might delay execution of finalizers, or not run them at all. This is allowed. See “How to coexist with the Garbage Collector” on page 23 for more details.

## Basic diagnostics (verbosegc)

A good way to see what is going on with garbage collection is to use verbosegc, which is enabled by the **-verbosegc** option. Note that verbosegc can, and usually does, change between releases.

## verbosegc output from a System.gc()

```
<GC(3): GC cycle started Tue Mar 19 08:24:34 2002
<GC(3): freed 58808 bytes, 27% free (1163016/4192768), in 14 ms>

<GC(3): mark: 13 ms, sweep: 1 ms, compact: 0 ms>
<GC(3): refs: soft 0 (age >= 32), weak 0, final 0, phantom 0>
```

The above verbosegc output is an example of a System.gc collection, or forced garbage collection. All the lines start with GC(3), indicating that this was the third garbage collection in this JVM. The first line shows the date and time of the start of the collection. The second line shows that 58808 bytes were freed in 14 ms, resulting in 27% free space in the heap. The figures in parentheses show the actual number of bytes that are free, and the total bytes that are available in the heap. The third line shows the times for the mark, sweep, and compaction phases. In this case, no compaction occurred, so the time is zero. The last line shows the reference objects that were found during this garbage collection, and the threshold for removing soft references. In this case, no reference objects were found.

## verbosegc output when pinnedFreeList is exhausted

```
<AF[5]: Allocation Failure. need 524 bytes, 31726137 ms since last AF>
<AF[5]: managing allocation failure, action=0 (496330792/536869376)>
<GC(612): GC cycle started Mon Mar 22 07:15:56 2004
<GC(612): freed 2213344 bytes, 92% free (498544136/536869376), in 66 ms>
<GC(612): mark: 54 ms, sweep: 12 ms, compact: 0 ms>
<GC(612): refs: soft 4 (age >= 32), weak 175, final 46, phantom 0>
<AF[5]: completed in 68 ms>
```

In release 1.4.1 and above, the Garbage Collector allocates a *pCluster* as the second object on the heap. A pCluster is an area of storage that is used to allocate any pinned objects. It is 16 KB long. A pinnedFreeList is also introduced. After every GC, an amount of storage is taken off the bottom of the free list and chained to the pinnedFreeList. Allocation requests for pClusters use the pinnedFreeList, while other allocation requests use the free list. When either free list is exhausted, an allocation failure and a GC occur. If the verbosegc output shows `action=0`, the pinnedFreeList was exhausted. For more details on pCluster, refer to "Pinned clusters" on page 12.

## verbosegc output from an allocation failure

```
<AF[5]: Allocation Failure. need 32 bytes, 286 ms since last AF>
<AF[5]: managing allocation failure, action=1 (0/6172496) (247968/248496)>
  <GC(6): GC cycle started Tue Mar 19 08:24:46 2002
  <GC(6): freed 1770544 bytes, 31% free (2018512/6420992), in 25 ms>
  <GC(6): mark: 23 ms, sweep: 2 ms, compact: 0 ms>
  <GC(6): refs: soft 1 (age >= 4), weak 0, final 0, phantom 0>
<AF[5]: completed in 26 ms>
```

The above verbosegc output is an example of an allocation failure (AF) collection. An allocation failure does not mean that an error has occurred in the code; it is the name that is given to the event that triggers when it is not possible to allocate a large enough chunk from the heap. The output contains the same four lines that are in the System.gc verbose output, and some additional lines. The lines that start with AF[5] are the allocation failure lines and indicate that this was the fifth AF collection in this JVM. The first line shows how many bytes were required by the allocation that had a failure, and how long it has been since the last AF. The second line shows what action the Garbage Collector is taking to solve the AF, and how much free space is available in the main part of the heap, and how much is available in the wilderness. The possible AF actions are:

**0** This action means that the Garbage Collector tried, but failed, to allocate from the pinned free list.

**1** This action performs a garbage collection without using the wilderness. It is designed to avoid compactions by keeping the wilderness available for a large allocation request.

**2** This action means that the Garbage Collector has tried to allocate out of the wilderness and failed.

**3** This action means that the Garbage Collector is going to attempt to expand the heap.

**4** This action means that the Garbage Collector is going to clear any remaining soft references. This occurs only if less than 12% free space is available in a fully-expanded heap.

**5** This action applies only to resettable mode and means that garbage collection is going to try to take some space from the transient heap.

**6** This is not an action. It outputs a verbosegc message to say that the JVM is very low on heap space, or totally out of heap space.

The last line shows how long the AF took. This includes the time it took to stop and start all the application threads.

## verbosegc output from a heap expansion

```
<AF[11]: Allocation Failure. need 24 bytes, 182 ms since last AF>
<AF[11]: managing allocation failure, action=1 (0/6382368) (10296/38624)>
  <GC(12): GC cycle started Tue Mar 19 08:24:49 2002
  <GC(12): freed 1877560 bytes, 29% free (1887856/6420992), in 21 ms>
  <GC(12): mark: 19 ms, sweep: 2 ms, compact: 0 ms>
  <GC(12): refs: soft 0 (age >= 32), weak 0, final 0, phantom 0>
<AF[11]: managing allocation failure, action=3 (1887856/6420992)>
  <GC(12): need to expand mark bits for 7600640-byte heap>
  <GC(12): expanded mark bits by 16384 to 118784 bytes>
  <GC(12): need to expand alloc bits for 7600640-byte heap>
  <GC(12): expanded alloc bits by 16384 to 118784 bytes>
  <GC(12): expanded heap by 1179648 to 7600640 bytes, 40% free>
<AF[11]: completed in 31 ms>
```

The above verbosegc output is an example of an AF collection that includes a heap expansion. The output is the same as a verbosegc output for an AF, with some additional lines for the expansion. It shows by how much the mark bits, the alloc bits, and the heap are expanded, and how much free space is available. In the example, the heap was expanded by 1179648 bytes, which gave 40% free space.

## verbosegc output from a heap shrinkage

```
<AF[9]: Allocation Failure. need 32 bytes, 92 ms since last AF>
<AF[9]: managing allocation failure, action=1 (0/22100560) (1163184/1163184)>
  <GC(9): may need to shrink mark bits for 22149632-byte heap>
  <GC(9): shrank mark bits to 348160>
  <GC(9): may need to shrink alloc bits for 22149632-byte heap>
  <GC(9): shrank alloc bits to 348160>
  <GC(9): shrank heap by 1114112 to 22149632 bytes, 79% free>
  <GC(9): GC cycle started Tue Mar 19 11:08:18 2002
  <GC(9): GC cycle started Tue Mar 19 11:08:18 2002
  <GC(9): mark: 4 ms, sweep: 3 ms, compact: 0 ms>
  <GC(9): refs: soft 0 (age >= 6), weak 0, final 0, phantom 0>
<AF[9]: completed in 8 ms>
```

This output is very similar to the verbosegc output for heap expansion. It shows by how much the mark bits, the alloc bits, and the heap are shrunk, and how much free space is available. In the example, the heap shrank by 1114112 bytes, resulting in 79% free space. One other difference between the verbosegc output for heap expansion and heap shrinkage is the sequence of the output. This difference is because expansion happens after all the threads have been restarted and shrinkage happens before all the threads have been restarted.

## verbosegc output from a compaction

```
<AF[2]: Allocation Failure. need 88 bytes, 5248 ms since last AF>
<AF[2]: managing allocation failure, action=1 (0/4032592) (160176/160176)>
  <GC(2): GC cycle started Tue Mar 19 11:32:28 2002
  <GC(2): freed 1165360 bytes, 31% free (1325536/4192768), in 63 ms>
  <GC(2): mark: 13 ms, sweep: 1 ms, compact: 49 ms>
  <GC(2): refs: soft 0 (age >= 32), weak 0, final 3, phantom 0>
  <GC(2): moved 32752 objects, 2511088 bytes, reason=2, used 8 more bytes>
<AF[2]: completed in 64 ms>
```

The above verbosegc example shows a compaction. The main difference between this and the outputs for expansion and shrinkage is the additional line that shows how many objects have been moved, how many bytes have been moved, the reason for the compaction, and how many additional bytes have been added. It is possible to have additional bytes because if the Garbage Collector moves an object that has been hashed, it has to store the hash value in the object. That action might mean increasing the object size. The possible reasons for a compaction are as follows:

- Following the mark and sweep phase, not enough free space is available for the allocation request.
- The heap is fragmented and will benefit from a compaction.
- Less than half the **-Xminf** value is free space (the default is 30% in which case this will be less than 15% free space), and the free space plus the dark matter is not less than **-Xminf**.
- A System.gc collection.
- Less than 5% free space is available.
- Less than 128 KB free space is available.
- The **-Xcompactgc** parameter has been specified.
- The transient heap has less than 5% free space available.
- A compaction is attempted before an attempt to shrink the heap.
- An incremental compaction is needed because of "dark matter".
- The **-Xpartialcompactgc** parameter has been specified.
- An incremental compaction is needed because of wilderness expansion.
- An incremental compaction is needed because not enough space is available for the wilderness.

## verbosegc output from a concurrent mark kickoff

```
<CONCURRENT GC Free= 379544 Expected free space= 378884 Kickoff=379406>
<Initial Trace rate is 8.01>
```

The above two lines are the verbosegc output that indicate that the concurrent phase has started. The first line shows how much free space is available, and how much will be available after this heap lock allocation. The Kickoff value is the level at which concurrent mark starts. In this example, the expected space is 378884, which is less than the Kickoff value of 379406. The second line shows the initial

trace rate. In this example, it is 8.01, which means that for every byte that is allocated in a heap lock allocation, the Garbage Collector must trace 8.01 bytes of live data.

## verbosegc output from a concurrent mark System.gc collection

```
<GC(23): Bytes Traced =0 (Foreground: 0+ Background: 0) State = 3 >
  <GC(23): GC cycle started Fri Oct 11 08:45:34 2002
  <GC(23): freed 12847376 bytes, 94% free (127145208/134216192), in 975 ms>
  <GC(23): mark: 408 ms, sweep: 70 ms, compact: 497 ms>
  <GC(23): refs: soft 0 (age >= 32), weak 0, final 0, phantom 0>
  <GC(23): moved 95811 objects, 6316896 bytes, reason=4>
```

Line 1 shows the state as a numeric value. The possible values for this field are:

- HALTED (0)
- EXHAUSTED (1)
- EXHAUSTED_BK_HELPER (2)
- ABORTED (3)

  In the example, it is 3, which means ABORTED, to show that concurrent mark did not complete the initialization phase and was therefore aborted.

Line 1 also shows the foreground and background trace values.

Line 2 shows the date and time of the start of the collection. Line 3 shows that 12847376 bytes were freed in 975 ms, resulting in 94% free space in the heap. The figures in parentheses show the actual number of bytes that are free, and the total bytes that are available in the heap. Line 4 shows the times for the mark, sweep, and compaction phases. Line 5 shows the reference objects that were found during this garbage collection, and the threshold for removing soft references. In this example, no reference objects were found. Line 6 shows the number of objects that were moved, the total size of those objects, and the reason why they were moved; in this example because of a system garbage collection.

## verbosegc output from a concurrent mark AF collection

```
<AF[7]: Allocation Failure. need 528 bytes, 493 ms since last AF or CON>
<AF[7]: managing allocation failure, action=1 (0/3983128) (209640/209640)>
<GC(8): Bytes Traced =670940 (Foreground: 73725+ Background: 597215) State = 0

  <GC(8): GC cycle started Tue Oct 08 13:43:14 2002
  <GC(8): freed 2926496 bytes, 74% free (3136136/4192768), in 8 ms>
  <GC(8): mark: 7 ms, sweep: 1 ms, compact: 0 ms>
  <GC(8): refs: soft 0 (age >= 32), weak 0, final 0, phantom 0>
<AF[7]: completed in 10 ms>
```

The above example shows an AF collection that has occurred while concurrent mark is running. This collection is for SDK 1.4.1.

The Traced figures in parentheses show how much is traced by the application threads and how much is traced by the background thread. The total bytes traced is the sum of the work done by the background and foreground traces. State is 0 (see above), which means that concurrent is HALTED.

## verbosegc output from a concurrent mark AF collection with :Xgccon

```
<AF[19]: Allocation Failure. need 65552 bytes, 106 ms since last AF or CON>
<AF[19]: managing allocation failure, action=1 (83624/16684008) (878104/878104)>
```

```
<GC(20): Bytes Traced =1882061 (Foreground: 1292013+ Background: 590048) State =0 >
  <GC(20): Card Cleaning Done. Cleaned:27 (0 skipped). Estimation 593 (Factor 0 .017)>
  <GC(20): GC cycle started Fri Oct 11 10:23:49 2002
  <GC(20): freed 8465280 bytes, 53% free (9427008/17562112), in 9 ms>
  <GC(20): mark: 7 ms, sweep: 2 ms, compact: 0 ms>
  <GC(20): In mark: Final dirty Cards scan: 41 cards
  <GC(20): refs: soft 0 (age >= 6), weak 0, final 0, phantom 0>
<AF[19]: completed in 13 ms>
```

The above is an example of an AF collection that has occured while concurrent mark is running with the :Xgccon parameter set. Line 3 shows a state of 0, which means that concurrent is HALTED. Line 4 shows that concurrent card cleaning was performed for 27 cards, while `estimation` is the number of dirty cards found.

## verbosegc output from a concurrent mark collection

```
<CON[41]: Concurrent collection, (284528/8238832) (17560/17168), 874 ms since last CON or AF>
<GC(45): Bytes Traced =5098693 (Foreground: 555297+ Background: 4543396) State =2 >
  <GC(45): GC cycle started Tue Oct 08 12:31:14 2002
  <GC(45): freed 2185000 bytes, 30% free (2487088/8256000), in 7 ms>
  <GC(45): mark: 5 ms, sweep: 2 ms, compact: 0 ms>
  <GC(45): refs: soft 0 (age >= 32), weak 0, final 0, phantom 0>
<CON[41]: completed in 9 ms>
```

The above verbosegc output is an example of a collection that is initiated by concurrent mark in SDK 1.4.1. It is very similar to the AF concurrent collection, except that instead of AF at the start of the lines, the trace is preceded by CON. In this example, the state is `EXHAUSTED_BK_HELPER`, which means that no more work was available for the background threads to do.

## verbosegc output from a concurrent mark collection with :Xgccon

```
<CON[20]: Concurrent collection, (397808/131070464) (3145728/3145728), 5933 ms since last CON or AF>
<GC(26): Bytes Traced =11845976 (Foreground: 4203037+ Background: 7642939) State= 1 >
  <GC(26): Card Cleaning Done. Cleaned:4127 (0 skipped). Estimation 3896 (Factor 0.015)>
  <GC(26): GC cycle started Fri Oct 11 09:45:32 2002
  <GC(26): wait for concurrent tracers: 1 ms>
  <GC(26): freed 117639824 bytes, 90% free (121183360/134216192), in 20 ms>
  <GC(26): mark: 10 ms, sweep: 10 ms, compact: 0 ms>
  <GC(26): In mark: Final dirty Cards scan: 838 cards
  <GC(26): refs: soft 0 (age >= 32), weak 0, final 0, phantom 0>
<CON[20]: completed in 21 ms>
```

The addition of the :Xgccon parameter to the verbose command returns the card cleaning information that was automatically generated in version 1.3.1. In this example, `state=1` means that no more work is available for concurrent mark to do. An extra line (line 5) has been added, to display the time that was spent waiting for concurrent tracers to complete.

## verbosegc output from resettable (z/OS only)

```
<TH_AF[8]: Transient heap Allocation Failure. need 64 bytes, 9716 ms since last TH_AF>
<TH_AF[8]: managing TH allocation failure, action=3 (0/4389888)>
  <GC(25): need to expand transient mark bits for 4586496-byte heap>
  <GC(25): expanded transient mark bits by 3072 to 71672 bytes>
  <GC(25): need to expand transient alloc bits for 4586496-byte heap>
  <GC(25): expanded transient alloc bits by 3072 to 71672 bytes>
  <GC(25): expanded transient heap fully by 196608 to 4586496 bytes>
<TH_AF[8]: completed in 1 ms>
```

When running resettable, the JVM has a middleware heap and a transient heap. The verbosegc for the transient heap is slightly different. In the above example, note the use of TH_AF instead of AF. The policy when running resettable is to expand the transient heap when an allocation failure occurs, instead of to garbage collect it. The above example shows a successful expansion. The example below shows what happens when the expansion is not successful. Here a garbage collection must be performed. The output shows how much space is freed from each of the heaps.

```
TH_AF[11]: Transient heap Allocation Failure. need 32 bytes, 16570 ms since last TH_AF>
<TH_AF[11]: managing TH allocation failure, action=3 (0/4586496)>
<TH_AF[11]: managing TH allocation failure, action=2 (0/4586496)>
  <GC(29): GC cycle started Tue Mar 19 14:47:42 2002
  <GC(29): freed 402552 bytes from Transient Heap 8% free (402552/4586496) and...>
  <GC(29): freed 1456 bytes, 38% free (623304/1636864), in 1285 ms>
  <GC(29): mark: 1263 ms, sweep: 22 ms, compact: 0 ms>
  <GC(29): refs: soft 0 (age >= 6), weak 0, final 0, phantom 0>
<TH_AF[11]: completed in 1287 ms>
```

Documentation for the JVMSet JVM can be found in *New IBM Technology featuring Persistent Reusable Java Virtual Machines*, SC34-6034-0. This is available at http://www.s390.ibm.com/Java

## Advanced diagnostics

The verbosegc option is the main diagnostic that is available for runtime analysis of the Garbage Collector. However, another set of command line options is available that can affect the behavior of the Garbage Collector, and that might aid diagnostics. These options are:

**-Xcompactexplicitgc**
**-Xdisableexplicitgc**
**-Xgcpolicy**
**-Xgcthreads**
**-Xnoclassgc**
**-Xnocompactgc**
**-Xnocompactexplicitgc**
**-Xnopartialcompactgc**

### -Xcompactexplicitgc

This option runs full compaction each time System.gc() is called. Its default behavior with a system.gc call is to perform a compaction only if an allocation failure triggered a garbage collection since the last system.gc call.

### -Xdisableexplicitgc

This option converts Java application call 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 some cases, these calls can degrade performance time 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 in production where testing had shown this to be beneficial; for example, from performance testing in conjunction with verbose:gc output. The new flag should not be set when one of the following is running:

- The zSeries JVM with CICS in resettable mode or with DB/2 stored procedures
- Performance profilers that make explicit garbage collection calls to detect object freeing and memory leaks
- Performance benchmarks in which explicit garbage collection calls are made between measurement intervals

## -Xgcpolicy:<optthruput | optavgpause | subpool>

Note that the subpool option is for AIX only.

When **-Xgcpolicy** is set to **optthruput**, concurrent mark is disabled. If you do not have pause time problems (as seen by erratic application response times) you should get the best throughput with this option. **Optthruput** is the default setting.

When **-Xgcpolicy** is set to **optavgpause**, concurrent mark is enabled with its default values. If you are having problems with erratic application response times that are caused by normal garbage collections, you can correct those problems, at the cost of some throughput, with the **optavgpause** option.

When **-Xgcpolicy** is set to **subpool** (introduced for AIX only in Version 1.4.1 Service Refresh 1), an improved algorithm for object allocation that aims to achieve better performance in allocating objects on the heap is used. This algorithm might provide additional throughput optimization because it tries to improve the efficiency of object allocation and reduce lock contention on large SMP systems. Concurrent mark is disabled when this policy is enabled.

## -Xgcthreads<n>

This option sets the number of threads (**<n>**) that are used by garbage collection for concurrent operations. The default is to use as many threads as there are processors. A reduction in the number of threads reduces concurrent operations, at the cost of performance, and might avoid potential problems in this area. No advantage is gained if you increase the number of threads above the default setting; you are strongly recommended not to do so.

## -Xnoclassgc

This option disables collection of class objects. Collecting class objects is a difficult operation. This option might expose problems in this area.

## -Xnocompactgc

This option disables heap compaction. If a problem exists in compaction, this option avoids that problem.

## -Xnocompactexplicitgc

This option never runs compaction when System.gc() is called. Its default behavior with a system.gc call is to perform a compaction only if an allocation failure triggered a garbage collection since the last system.gc call.

### -Xnopartialcompactgc

Never run an incremental compaction. The default is ″false″; that is, incremental compaction is enabled.

## Tracing

This section describes the garbage collection trace facilities. For instructions about how to activate these traces, see Chapter 33, “Tracing Java applications and the JVM,” on page 321.

The most exhaustive garbage collection diagnostic is garbage collection trace. Take care what you choose to trace because enormous amounts of trace data can be generated. From JVM version 1.4.1, garbage collection trace options (earlier known as tracegc) have been merged with the JVM RAS trace. Therefore, although the information that is available from this trace has remained mostly unchanged, usage of the trace has changed. Table 23 compares the tracegc options that are available in version 1.4 with the new ST trace options that are provided from JVM 1.4.1.

*Table 23. Comparison of tracegc options*

| Old tracegc option | Corresponding JVM trace option |
|---|---|
| TRACEGC_TERSE | st_terse |
| TRACEGC_VERIFY | st_verify |
| TRACEGC_MARK | st_mark |
| TRACEGC_COMPACT | st_compact |
| TRACEGC_COMPACT_VERBOSE | st_compact_verbose |
| TRACEGC_COMPACT_DUMP | st_compact_dump |
| TRACEGC_DUMP | st_dump |
| TRACEGC_ALLOC | st_alloc |
| TRACEGC_REFS | st_refs |
| TRACEGC_BACKTRACE | st_backtrace |
| TRACEGC_LOGFILE | Not applicable |
| TRACEGC_FREELIST | st_freelist |
| TRACEGC_CALLOC | st_calloc |
| TRACEGC_PARALLEL | st_parallel |
| TRACEGC_TRACE | st_trace |
| TRACEGC_CONCURRENT | st_concurrent |
| TRACEGC_CONCURRENT_PCK | st_concurrent_pck |
| TRACEGC_ICOMPACT | st_icompact |
| TRACEGC_CONCURRENT_SHADOW_HEAP | st_concurrent_shadow_heap |

TRACEGC_LOGFILE is not implemented in version 1.4.2, because JVM trace uses **-Dibm.dg.trc.outputto** to redirect trace outputs to a file. Also, IBM_JVM_ST_VERIFYHEAP, which checks the integrity of the heap, has been replaced with **-Dibm.dg.trc.print=st_verify_heap**.

To activate the TRACEGC_MARK trace, for example, use **-Dibm.dg.trc.print=st_mark**. If you want to activate more than one trace, add the

identifiers to the command line together. For example, if you want the TRACEGC_ALLOC and the TRACEGC_REFS, use **-Dibm.dg.trc.print=st_alloc,st_refs**.

## st_terse

This trace dumps the contents of the heap before and after a garbage collection. If running with the -Xresettable option, the transient heap is also dumped. The system heaps are not dumped. This can be a very large trace.

```
*DH(1)* 0x10063064 a x0x58 java.lang.Thread@10063068/10063070
*DH(1)* 0x100630bc a x0x58 java.lang.ref.Finalizer$FinalizerThread@100630C0/100630C8
*DH(1)* 0x10063114 a x0x58 java.lang.ref.Reference$ReferenceHandler@10063118/10063120
*DH(1)* 0x1006316c a x0x58 java.lang.Thread@10063170/10063178
*DH(1)* 0x100631c4 a x0x58 java.lang.Thread@100631C8/100631D0
*DH(1)* 0x1006321c f x0x28
*DH(1)* 0x10063244 a x0x50 java.util.HashMap$Entry[16]
*DH(1)* 0x10063294 a x0x20 java.lang.String@10063298/100632A0
*DH(1)* 0x100632b4 a x0x18 "float"
*DH(1)* 0x100632cc a x0x30 java.util.HashMap@100632D0/100632D8
*DH(1)* 0x100632fc a x0x10 java.lang.Object@10063300/10063308
*DH(1)* 0x1006330c a x0x10 java.lang.ref.ReferenceQueue$Lock@10063310/10063318
*DH(1)* 0x1006331c a x0x20 java.lang.ref.ReferenceQueue@10063320/10063328
*DH(1)* 0x1006333c a x0x10 java.lang.ref.ReferenceQueue$Lock@10063340/10063348
*DH(1)* 0x1006334c a x0x20 java.lang.ref.ReferenceQueue$Null@10063350/10063358
*DH(1)* 0x1006336c a x0x10 java.lang.ref.ReferenceQueue$Lock@10063370/10063378
*DH(1)* 0x1006337c a x0x20 java.lang.ref.ReferenceQueue$Null@10063380/10063388
```

The above example shows a small part of this trace. Each line is either an object or a free chunk. The first field indicates that this is a heap dump by the letters DH. The number in parentheses is the garbage collection number. The next field is the address of the object followed by an indication of whether this is an allocated object or a free chunk, **a** for allocated and **f** for free. The next field is the length of the object. The final field is the class of the object, or blank if it is a free chunk.

## st_verify

This trace verifies the integrity of the heap before and after a compaction. Some messages are displayed for error conditions, but most of the checking is done via sysAsserts, so therefore this trace should be run with the debug build. If you run this with the -Xresettable option, the transient heap is also verified. This is a small trace.

```
GC(VFY-SUM): pinned=79(classes=2/freeclasses=0) dosed=85 movable=4609 free=1886
GC(VFY-SUM): freeblocks: max=37080 ave=184 (347384/1886)
GC(VFYAC-SUM): freeblocks: max=94504 ave=7389 (347328/47)
```

The above example shows the output from a healthy heap.

The first line displays the state before a compaction. In this example are 79 pinned objects, of which two are classes. The freeclasses field is not used and is always zero. The example also has 85 dosed objects. The movable field is the count of all objects that are not pinned or dosed. The free field shows the number of free chunks.

The second line also shows the state before a compaction. The freeblocks field shows the maximum and average sizes of the free blocks. The numbers in parentheses are the total amount of free space and the number of free chunks.

The third line shows the state after a compaction. The freeblocks field shows the maximum and average size of the free blocks. The numbers in parenthesis are the total amount of free space and the number of free chunks.

## st_mark

This trace traces all the objects that are found during the conservative part of marking. This is a small to medium sized trace.

```
1--> pinned(jh) allocator@10061F50/10061F58
1J> jframe( 822f48c)
1--> pinned(ch) x0xb99ff884 java.lang.String@103A6460/103A6468
cch java.lang.String@103A6460/103A6468
1--> pinned(ch) x0xb99ff888 java.io.PrintStream@10074770/10074778
```

The above example shows a small section of this trace.

The first line is displayed for a pinned object, whether it is going to be marked or not. The number at the start is the garbage collection cycle number. The letters (*jh*) show that this pinned object was found as a reference to a handle on a Java frame. Other possibilities are:

- ch: A reference to a handle on a C frame.
- co: A reference to an object on a C frame.

The last field is the name of the object.

The second line is displayed when a Java frame is traced. The address of the Java frame is displayed.

The third line shows another pinned object, this time found as a reference to a handle on a C frame. In this case, the address of the object is displayed in addition to the name.

The fourth line is displayed for objects that are about to be marked. The letters *cch* show that this object was found as a reference to a handle on a C frame. Other possibilities are:

- cch: A reference to a handle on a C frame.
- cco: A reference to an object on a C frame.

The last field is the name of the object.

## st_compact

This trace traces all the moves that occur during a compaction. Virtual moves are traced from reverseHandlesAndUpdateForwardRefs() and actual moves are traced from MoveObjectsAndUpdateBackwardRefs(). If running with the -Xresettable option, the transient-heap moves are also traced. This can be a large trace.

```
<GC(1): to move(slide) 0x100674a4 (x0x30) to 0x10067364 (x0x30) java.util.Hashtable@100674A8/100674B0>
<GC(1): to move(slide) 0x100674d4 (x0x20) to 0x10067394 (x0x20) java.util.Hashtable$Entry[4]>
```

The above example shows a small part of the trace of the virtual move. The first field is the garbage collection cycle number. The word (*slide*) shows that this object is to be slid down the heap so that it is moved down to be adjacent to another object. The other possibility is (*lift*), which would show that this object is to be moved to a free space that is lower down the heap. The next field shows where the object is, to where the object is going to move, and the size of the object. The last field is the name of the object.

```
<GC(1): moving(slide) 0x10063244 (x0x50) to 0x1006321c (x0x50) java.util.HashMap$Entry[16]>
<GC(1): moving(slide) 0x10063294 (x0x20) to 0x1006326c (x0x20) java.lang.String@10063270/10063278>
```

The above example shows a small part of the trace of the actual move. The only difference between this and the virtual trace move is the word *moving* instead of the words *to move.*

## st_compact_verbose

This trace can be run only with st_verify. Therefore, you invoke it by using -Dibm.dg.trc.print=st_verify,st_compact_verbose. It shows all the pinned and dosed objects that are on the heap before a compaction. This is a small to medium sized trace.

```
<GC(VFY): pinned java.lang.Thread@10063068/10063070>
<GC(VFY): pinned java.lang.ref.Finalizer$FinalizerThread@100630C0/100630C8>
<GC(VFY): pinned java.lang.ref.Reference$ReferenceHandler@10063118/10063120>
<GC(VFY): pinned java.lang.Thread@10063170/10063178>
<GC(VFY): dosed  java.lang.ref.ReferenceQueue$Null@10063300/10063308>
<GC(VFY): dosed  java.lang.ref.ReferenceQueue$Lock@1006EE78/1006EE80>
<GC(VFY): dosed  java.lang.ref.ReferenceQueue@1006EE88/1006EE90>
```

The above example shows a portion of this trace. Each pinned or dosed object is displayed along with its name.

## st_compact_dump

This trace can be run only with st_verify. Therefore, you use -Dibm.dg.trc.print=st_verify,st_compact_dump to invoke it. It dumps the contents of the heap before a compaction and after a compaction. This can be a very large trace.

The format of the output is the same as for st_terse.

## st_dump

This trace can be run only with st_verify. Therefore, you use -Dibm.dg.trc.print=st_verify,st_dump to invoke it. It displays every free chunk that is in the heap, including dark matter (free chunks that are too small to be on the free list). If it runs with the -Xresettable option, the transient-heap free chunks are also displayed. This is a medium sized trace.

```
<GC(75) Dumping Middleware Heap free blocks
<GC(75) 0x1006f13c freelen=x0x8 -- x0x10 java.lang.ref.Reference$Lock@1006F140/1006F148
<GC(75) 0x100cf334 freelen=x0x29b30 -- x0xfb0 [Lgt;[1000]
<GC(75) 0x103c5134 freelen=x0x2f2c00 p- x0x20 byte[][20]
<GC(75) 0x103cb434 freelen=x0x6280 -- x0x60 gt[20]
<GC(75) 0x103ffbfc freelen=x0x332e8
```

The above example shows part of this trace. The first field is the garbage collection cycle number. The next two fields show the address of the free chunk and its length. The next field shows information about the live object that follows the free chunk. In the example are the characters "—" (dashes). These are replaced by **p** and **d** if the object is pinned or dosed. The last field shows the length and name of the live object that follows the free chunk. In this example, the last free chunk does not have a live object after it.

## st_alloc

This trace traces every heap lock allocation. This is a large trace.

```
*77* alc-mwo 0x10383948 java.lang.String
*77* alc 0x40516508 *ClassClass*
*77* alc-tma 10383908 5[24]
```

```
*77* alc-mwo 0x103838e8 java.lang.String
*77* alc-mwo 0x103838d8 java.util.HashMap$KeySet
*77* alc 0x40516638 *ClassClass*
*77* alc-tma 10383890 5[29]
```

The above example shows a small portion of this trace. The first field shows the number of garbage collection cycles that have taken place so far. The functions that can issue the allocation are:

- alc
- allocSystemApplicationClass
- allocSystemClass
- allocSystemPrimitiveArray
- allocSystemStringObject
- clonePrimitiveArrayToSystemHeap
- allocTransientClass
- allocTransientArray
- alc-mwc - allocMiddlewareClass
- alc-tma - targetedAllocMiddlewareArray
- alc-mwo - allocMiddlewareObject
- alc-mca - allocMiddlewareContextArray
- alc-mco - allocMiddlewareContextObject
- alc-pba - allocatePinnedByteArray
- alc-arr - allocArray
- alc-obj - allocObject
- alc-cxa - allocContextArray
- alc-cob - allocContextObject
- alc-ash - allocArrayInSameHeap

The address of the allocated object is displayed after the function. Following this, one of these is displayed:

- If it is an array, the type code of the array and the number of elements in the array (in square brackets) are displayed.
- If it is a class block, *ClassClass* is displayed.
- If it is a normal object, the object name is displayed.

## st_refs

This trace traces reference handling during garbage collection. The size of this trace depends on how many reference objects are found.

```
ref java.lang.ref.SoftReference@10084838/10084840
     -> java.lang.reflect.Constructor[1]
  skip java.lang.reflect.Constructor[1]
ref java.lang.ref.SoftReference@10095398/100953A0
     -> java.lang.reflect.Constructor[1]
  skip java.lang.reflect.Constructor[1]
ref java.lang.ref.SoftReference@10092960/10092968
    -> java.lang.reflect.Constructor[1]
  skip java.lang.reflect.Constructor[1]
processRefList: Dropping java.lang.ref.WeakReference@10085E90/10085E98
processRefList: Dropping java.lang.ref.Finalizer@100966E8/100966F0
processRefList: Dropping java.lang.ref.Finalizer@1009A460/1009A468
processRefList: Enqueuing java.lang.ref.Finalizer@10085740/10085748
processRefList: Enqueuing java.lang.ref.Finalizer@100D2988/100D2990
```

The above example shows three reference objects being processed.

The first part of the traces shows the references that are found during the marking phase. For each reference found, three lines are displayed (wrapped in the above example). The first line, starting with *ref*, together with the second line shows the reference object, and the third line, starting with *skip*, shows the referent.

The second part of the trace shows how each of these references is processed after marking has completed. If the referent is marked, the reference is dropped, as indicated by the word *Dropping*. If the referent is not marked, the reference is enqueued for further processing, as indicated by the word *Enqueuing*.

## st_backtrace

This trace can be run only with st_terse. Therefore, you use -Dibm.dg.trc.print=st_backtrace,st_terse to invoke it. It adds one line to the beginning of the st_terse trace, an example of which follows:

```
"Thread-1"(0x8274030)
```

This shows the name of the thread that is running garbage collection, followed by the address of the sys_thread_t structure.

## st_freelist

This trace traces information about the freelist during garbage collection. This is a small trace.

```
*1* free:    0   deferred:    1   total:     1
Alloc TLH: count 93, size 35352, discard 1752
non-TLH: count 0, search 0, size 0, discard 0
```

The above example is the output from one garbage collection cycle. The number that is displayed as **1** is the garbage collection cycle number.

In the first line, the `free` field shows the number of free chunks that are on the freelist at the beginning of a garbage collection cycle. The `deferred` field shows the number of free chunks that are on the deferredlist at the beginning of a garbage collection cycle. The `total` field shows the total number of free chunks that are on both lists.

The second line shows the activity of TLH allocations since the last garbage collection. The `count` field shows the number of TLH allocations. The `size` field shows the average size of TLH allocated. The `discard` field shows the amount of space that is discarded when a TLH is completed.

The third line shows the activity of heap lock allocations since the last garbage collection. The `count` field shows the number of heap lock allocations. The `search` field shows the average number of free chunks that had to be searched before one that was big enough was found. The `size` field shows the average size of heap lock allocations. The discard field shows the amount of space that was discarded because it was less than the minimum free chunk size.

## st_calloc

This trace traces successful calls to realObjCalloc() and transientRealObjCalloc(). This is a small trace. It generates some traces very early in the JVM initialization

process, at which time the tracing thread is not yet initialized. Therefore, to get all the information from this trace, you must pass the -Dibm.dg.trc.initialization option to Java.

```
<GC(0): tried to calloc, java.lang.Thread(0x0x100631c8:72)>
<GC(0): tried to calloc, java.lang.Thread(0x0x10063170:72)>
<GC(0): tried to calloc, java.lang.ref.Reference$ReferenceHandler(0x0x10063118:72)>
<GC(0): tried to calloc, java.lang.ref.Finalizer$FinalizerThread(0x0x100630c0:72)>
<GC(0): tried to calloc, java.lang.Thread(0x0x10063068:72)>
<GC(0): tried to calloc, java.lang.Thread(0x0x10063010:72)>
<GC(0): tried to calloc, java.util.logging.LogManager$Cleaner(0x0x10062fb8:76)>
<GC(0): tried to calloc, sun.misc.Launcher$ExtClassLoader(0x0x10062f40:108)>
<GC(0): tried to calloc, sun.misc.Launcher$AppClassLoader(0x0x10062ec8:104>
<GC(0): tried to calloc, java.lang.Thread(0x0x10062e70:72)>
<GC(0): tried to calloc, gc5(0x0x10062e18:72)>
<GC(0): tried to calloc, gc5(0x0x10062dc0:72)>
<GC(0): tried to calloc, java.lang.Thread(0x0x10062d68:72)>
```

The above example shows the output from this trace. For every successful call to one of the calloc() functions, a line is displayed that shows the name of the object, the address at which the object was allocated, and the size of the object.

## st_parallel

This trace traces the activity of parallel mark and parallel sweep, producing some statistics. Invoke it by using -verbosegc. This is a small trace.

```
Mark:   busy  stall  tail   ---publish---    ----steal---    --withdraw--
   0:            2       0     0     298/      1           0/      0      49/      1
   1:            2       0     0         0/      0       249/      2        0/      0
Sweep:  busy   idle sections 64  merge 0
   0:             0       1         31
   1:             0       1         33
```

The above example shows the output from a garbage collection cycle. It has one master thread and one helper thread. All times are in milliseconds.

The statistics for parallel mark are displayed first:

**Mark** The identification of the thread (the master thread will be 0).

**busy** The time that was spent scanning references.

**stall** The amount of time that was spent without work (either waiting for the other threads to finish or waiting for data to steal).

**tail** The time that it took to terminate when all the threads have completed.

**publish**
The total number of references that were moved to the Mark Queue, and the number of times references were moved to the Mark Queue.

**steal** The total number of references that were taken off the Mark Queues of other threads, and the number of times references were taken off the Mark Queues of other threads.

**withdraw**
The total number of references that were removed from the Mark Queue of the thread, and the number of times references were removed from the Mark Queue of the thread.

Next, the statistics for parallel sweep are displayed. The first line displays how many sections exist and the time it took to merge the sections:

**Sweep**
The identification of the thread (the master thread will be 0).

**busy** The time that was spent sweeping sections.

**idle** The amount of time that was spent without work.

**sections**
The number of sections that were swept.

**merge** The amount of time that was spent merging sections.

## st_trace

This trace traces all the work that is done by the mark phase. This is a very large trace.

```
****** Starting trace for GC ******
0x8274acc  Scanning Clusters
0x8274acc      0x10081ae0 =java.lang.ref.Finalizer@10081AE0/10081AE8
0x8274acc      0x1006f140 =java.lang.ref.Reference$Lock@1006F140/1006F148
0x8274acc      0x100d0638 = java.io.FileDescriptor@100D0638/100D0640
...
...
...

0x8274acc  Scanning Threads
0x8274acc   scan thread 0x8051388...
0x813ad74      0x100841b8 = java.lang.String@100841B8/100841C0
0x8274acc   scan saved registers...
0x813ad74      0x10084550 = java.lang.String@10084550/10084558
0x8274acc   scan native stack...
0x813ad74      0x10074738 = java.lang.String@10074738/10074740
0x8274acc   scan Java stack...
0x813ad74      0x100871c8 = java.lang.String@100871C8/100871D0
0x8274acc   .. get exception object...
0x813ad74      0x100844f8 = java.lang.String@100844F8/10084500
0x8274acc   .. get pending async exception...
```

The above example parts of one of these traces.

The first field is the address of the ExecEnv of the thread that is doing the marking.

At the start, all the roots are gathered; in this case, only one root from scanning threads. For each object that is pushed to the mark stack, the address is displayed, followed by an equal sign, followed by the name.

Then, local marking is started by popping an object, scanning it, and pushing any references that are found. For each object that is popped, the address is displayed, followed by a comma, followed by the name. Then, all pushed objects are displayed, as were pushed root objects. All pushed objects are indented.

## st_concurrent

This trace traces the state of WorkPackets that are in concurrent mark. In nondebug mode, this is a small trace. In debug mode, this is a large trace. Data is displayed as follows:

- When a background thread is activated after a start of concurrent collection. This requires -verbosegc. Here is an example:

  ```
  <CONCURRENT GC BK thread 0x00d45678  activated after GC(7)>
  ```

- When a background thread returns to waiting for the next concurrent garbage collection. This requires -verbosegc. Here is an example:

  ```
  <CONCURRENT GC BK thread x00d45678 (started after GC(7)) traced  25678>
  ```

- When concurrent collection performs a small stop-all-threads, to scan all the stacks of threads that have not yet scanned their own stacks. Here is an example:

  ```
  < 0x00d45678 scanned 5 stacks (30 -> 35)  trace total=25678>
  ```

  – 0x00d45678 is the ExecEnv pointer of the thread that is initiating this stop-all-threads.
  – It scanned 5 stacks.
  – 35 stacks are now scanned.
  – 25678 bytes were traces by the concurrent collection.
  – 30 threads have scanned their own stacks before this.

- When the attempt to stop-all-thread above failed. Here is an example:

  ```
  < failed to suspend threads for stacks scan>
  ```

In addition, **st_concurrent** traces various types of operations that were done on WorkPackets while it was performing the concurrent collection. These messages differ in the starting and ending symbol. They do not appear in separate lines, because many of them are expected. This requires the debug build.

The following trace is done when WorkPackets are reset at the end of the final STW phase:

```
#msg,next_ptr , packet_ptr,packet_mode#
```

- *msg* can be all kinds of headers, describing the list from which the packets are returned.
- *next_ptr* is the logical pointer user to point to next packet, shifted left 8 (to save screen space).
- *packet_ptr* is the physical pointer of the packet.
- *packet_mode* is the mode of the packet.

This trace should happen once for each WorkPacket, for each garbage collection.

The following trace should happen very rarely; that is, when a referent (that was valid when this object was linked into the reference object's list) becomes NULL:

```
?? Found NULL referent,ee,pkt_ptr,pk_mode (reference_obj_ptr->referent_ptr)??
```

- *ee* is the ExecEnv, shifted left 8 (to save screen space).
- *pkt_ptr* is the physical pointer of the packet.
- *pkt_mode* is the mode of the packet
- *reference_obj_ptr* is the reference object.
- *referent_ptr* is the referent.

## st_concurrent_pck

This trace traces frequent operations on WorkPackets, such as getting, or returning. It is an extremely large trace. Here is the format of each trace item:

```
[ msg, ee, packet_ptr, packet_mode ]
```

- *msg* can be various states of get or put operation (for example, GRF represents "Get Relatively Full", GN represents "Get Non Empty").
- *ee* is the ExecEnv, shifted left 8 (to save screen space).
- *packet_ptr* is the pointer of the packet.

- *packet_mode* is the desired mode of the packet (in case of get) or actual mode (in case of put).

## st_icompact

This trace traces, by an incremental compaction, all the work that has been done. This is a small trace. It generates some traces very early in the JVM initialization process, at which time the tracing thread is not yet initialized. Therefore, to get all the information from this trace, you must pass the -Dibm.dg.trc.initialization option to Java. Here is an example trace:

```
ICOMPACT Compaction region size is 33554432 bytes
ICOMPACT INITIAL er_log_area_size 25

ICOMPACT Started. Compaction region is now from 0x10000200 to 0x12000000.
         ICOMPACT Before trace 594 objects in Compaction Region
ICOMPACT-HEAP Started with 4410 objects in Compaction Region after trace
ICOMPACT Section 0 starts at 0x10000200 and ends at 0x11000100 (16383KB)
ICOMPACT Section 1 starts at 0x11000100 and ends at 0x12000000 (16383KB)
ICOMPACT-HEAP Thread 1 icBuildAllBreakTables begins
ICOMPACT Thread 0 used 758 entries in break table
ICOMPACT Thread 0 buildBreakTableForOneSection: 0 ms
ICOMPACT Thread 1: Empty section. Nothing done to break table
ICOMPACT Thread 1 buildBreakTableForOneSection: 2 ms
ICOMPACT-HEAP Thread 1 icBuildAllBreakTables complete: 2 ms
ICOMPACT-HEAP Thread 1 icFixup begins
ICOMPACT Thread 0 icFixUpClasses: 14 ms
ICOMPACT Thread 1 icFixUpClasses: 1 ms
ICOMPACT Thread 1 icFixUpRootRefs: 1 ms
ICOMPACT Thread 1 icFixUpClasses: 0 ms
ICOMPACT-HEAP Thread 1 icFixUp complete: 16 ms
ICOMPACT-HEAP Thread 1 icMoveAllObjects begins
ICOMPACT Thread 0 moveObjectsInOneSection: 2 ms
ICOMPACT-HEAP Thread 1 icMoveAllObjects complete: 3 ms
ICOMPACT-HEAP complete in 28 ms
ICOMPACT Started. Compaction region is now from 0x12000000 to 0x14000000.
         ICOMPACT Before trace 3 objects in Compaction Region
```

The example shows the start of one of these traces. The actions that are displayed in the first paragraph are performed only once.

The first line output shows the size of a compaction region during this run of incremental compaction. The second line shows the log value of the compaction area. This value is used in later calculations. This example output has one main thread (1) and one helper thread (0).

The boundaries of the first region that are to be compacted are displayed. The following lines show how many objects are in the area that is to be compacted. The number of objects is calculated by the number of allocbits that are on in the region, firstly before the mark phase has run, then before the incremental compaction is performed. These two calculations show whether the number of objects in the compaction region is increasing or decreasing before the compaction is performed. The compaction region is divided by the number of active threads, and a line is output to show each section.

Incremental compaction builds a break table to denote the blocks of active objects that are in the region. In this case, thread 0 has built a table with 758 blocks and taken 0 milliseconds to do so. Thread 1 has found an empty section. The main thread then summarizes the time that was taken to complete the break table.

The FixUp phase fixes all references in the system that point to the compaction region. GCHelper threads are used to aid in the identification of heap references to objects that are in the compaction region. These references must be fixed when the objects are moved by the compaction phase. The main thread does the fixup of all the roots. This fixup scans all nonsystem classes and performs all needed nonconservative marking. It then fixes references from the root set references to moved objects. Finally it helps with the fixup of heap objects in parallelFRFixup and a summary of total time take for the fixup phase is output.

Now that it has been determined what can be moved and where, it remains only to actually move the blocks of objects. Each thread moves the objects in its own section, and the main thread then issues a summary message. Finally, a summary message for this iteration of incremental compaction is issued, and the boundaries of the next incremental compaction region are displayed.

### st_concurrent_shadow_heap

To use this trace, you must also be running the debug build. You can find information about the use of the Shadow Heap in the Java2Wave2 database at 'ST Team room/Service Transfer and other education/Shadow Heap and Stored Card Table'.

## Heap and native memory use by the JVM

The JVM itself makes little use of the heap except for class objects. Class objects also use native memory.

The JVM does use native memory, but, for efficiency, does not use standard stack frames. The JIT (see Chapter 4, "Understanding the JIT," on page 37), the MMI (see Chapter 30, "JIT diagnostics," on page 295), 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 29, "Using the dump formatter," on page 261). 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, 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 32. 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 320

## Class-loader command-line options

These extended command-line options are available:

**-Xverify**
: This option enforces strict class-loading checks on classes that are loaded by way of the extensions and application class loaders. The default is that strict checking is not performed.

**-Xverify:none**
: This option disables strict class-loading checks on all class loaders. The default is that strict checks are enforced except on the JVM internal class loaders.

**-Xverify:remote**
: This option 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:\tests>java -Dibm.cl.verbose=Dick Tom
```

might produce output that is similar to this:

```
ExtClassLoader attempting to find Dick
ExtClassLoader using classpath D:\jre\lib\ext\gskikm.jar;D:\jre\lib\ext\ibmjceprovider.jar;
  D:\jre\lib\ext\indicim.jar;D:\jre\lib\ext\jaccess.jar;D:\jre\lib\ext\ldapsec.jar;
  D:\jre\lib\ext\oldcertpath.jar
ExtClassLoader could not find Dick.class in D:\jre\lib\ext\gskikm.jar
ExtClassLoader could not find Dick.class in D:\jre\lib\ext\ibmjceprovider.jar
ExtClassLoader could not find Dick.class in D:\jre\lib\ext\indicim.jar
ExtClassLoader could not find Dick.class in D:\jre\lib\ext\jaccess.jar
ExtClassLoader could not find Dick.class in D:\jre\lib\ext\ldapsec.jar
ExtClassLoader could not find Dick.class in D:\jre\lib\ext\oldcertpath.jar
ExtClassLoader could not find Dick

AppClassLoader attempting to find Dick
AppClassLoader using classpath C:\tests;C:\tests;D:\lib\tools.jar
AppClassLoader could not find Dick.class in C:\tests
AppClassLoader could not find Dick.class in D:\lib\tools.jar
AppClassLoader could not find Dick

Exception in thread "main" java.lang.NoClassDefFoundError: Dick
        at Tom.main(Tom.java:6)
```

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 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 trace. You can enable tracepoints at runtime by using levels, components, group names, or individual tracepoint identifiers.

This chapter describes JVM trace in:



The trace tool provides an extremely powerful ability to diagnose the JVM. It is simple to understand and simple to use effectively.

## 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 **ibm.dg.trc.methods** property, you can select method trace by class, method name, or both. Wildcards can be used, and a not operator is provided to allow for complex selection criteria. Note that this property selects only the methods that are to be traced. The MT trace component must be selected for a given trace destination. For example:

```
-Dibm.dg.trc.methods=*.*,!java/lang/*.*
-Dibm.dg.trc.print=mt
```

This routes method trace to stderr for all methods for all classes except those that start with java/lang. On some platforms, input parameters and return values can also be traced provided that the JIT is disabled.

### 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.

## Internal trace

IBM JVMs are 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.

**Note:** No guarantee is given that tracepoints will not vary from release to release and from platform to platform.

# 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 the **ibm.dg.trc.buffers** system property 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-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 Snap*nnnn.yyyymmdd.hhmmssth.process*.trc, where *nnnn* is a sequence number starting at 0001 (at JVM startup), *yyyymmdd* is the current date, *hhmmssth* 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 allows the thread to continue to run while a full trace buffer is written to disk. Depending on trace volume, buffer size, and the bandwidth of the output device, multiple buffers might be allocated to a given thread to keep pace with trace data that is being generated.

A thread is never stopped to allow trace buffers to be written. If the rate of trace data generation greatly exceeds the speed of the output device, excessive memory usage might occur and cause out-of-memory conditions. To prevent this, use the **nodynamic** option of the **ibm.dg.trc.buffers** system property. For long running trace runs, a wrap option is available to limit the file to a given size. See the **ibm.dg.trc.output** property 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 dumps 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 28, “Using method trace,” on page 257.

### 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:

- Through system properties at startup
- 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 is disabled and cannot be enabled later in the same run. To use trace, you must specify at least one trace system property at startup. If you have done this, you can then control trace by using various mechanisms later in the run. Note that by specifying unresettable event logging, you also enable trace.
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 property descriptions" on page 326 and see the appropriate trace property description for more information.

## Specifying trace system properties

The primary way to control trace is through system properties that you specify on the launcher command line or in the IBM_JAVA_OPTIONS environment variable. Some trace system properties are of the form `system.property.name`, while others are of the form `system.property.name=value`, where `system.property.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.

The syntax for specifying system properties depends on the launcher. Usually, it is:

```
java -Dsystem.property.name -Danother.property.name=value HelloWorld
```

but for some launchers, it might be:

```
javac -J-Dsystem.property.name -J-Danother.property.name=value HelloWorld.java
```

Depending on the platform and command line shell, properties that contain special characters must be enclosed in double quotes, as shown here:

```
java -D"property.with.special.characters=x(y)" HelloWorld
```

When you use the IBM_JAVA_OPTIONS environment variable, use this syntax:

```
set IBM_JAVA_OPTIONS=-Dsystem.property.name -Danother.property.name=value
```

or

```
export IBM_JAVA_OPTIONS=-Dsystem.property.name -Danother.property.name=value
```

**Note:** The JVM ignores misspelled properties. For this reason, the trace facility echoes the properties that it has found during startup, in the following way:

```
JVMDG200: Diagnostics system property ibm.dg.trc.buffers=20k
JVMDG200: Diagnostics system property ibm.dg.trc.print=dg
```

It is worthwhile looking at the stderr output stream to check whether the requested options have been recognized. Note that the content of a properties file is not echoed, but any errors that are in the file are indicated, and the JVM fails to initialize.

## Trace property summary

This section describes:



### Properties that control tracepoint selection

These properties enable and disable tracepoints. They also determine the destination for the trace data. In some cases, you must use them with other properties. 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 property to specify the destination filename.

These properties have equivalents in the Java and JVMRI API that was mentioned earlier.

*Table 24. Properties that control tracepoint selection*

| | |
|---|---|
| **ibm.dg.trc.minimal** | Trace selected tracepoints (identifier and timestamp only) to in-core buffer. Associated trace data is not recorded. |
| **ibm.dg.trc.maximal** | Trace selected tracepoints (identifier and timestamp and associated data) to in-core buffer. |
| **ibm.dg.trc.count** | Count the number of times selected tracepoints are called in the life of the JVM. |
| **ibm.dg.trc.print** | Trace selected tracepoints to stderr with no indentation. |
| **ibm.dg.trc.iprint** | Trace selected tracepoints to stderr with indentation. |
| **ibm.dg.trc.platform** | Route selected tracepoints to the platform trace engine (z/OS only). |
| **ibm.dg.trc.external** | Route selected tracepoints to a JVMRI listener. |
| **ibm.dg.trc.exception** | Trace selected tracepoints to an in-core buffer reserved for exceptions. |

## Properties that indirectly affect tracepoint selection

These properties affect the availability of particular tracepoints but unless you specify them with a tracepoint selection property, they have no effect other than possibly degraded performance. For example, if you specify the initialization property, tracing occurs during JVM startup, but you must activate the actual tracepoints by using a tracepoint selection property.

*Table 25. Properties that indirectly affect tracepoint selection*

| | |
|---|---|
| **ibm.dg.trc.methods** | Select classes and methods to trace. |
| **ibm.dg.trc.highuse** | Select components for which high use trace data is to be collected. |
| **ibm.dg.trc.initialization** | Enable trace during JVM initialization. |
| **ibm.dg.trc.applids** | Select Java applications that are instrumented for application trace. |

## Triggering and suspend or resume

These system properties provide mechanisms to tailor trace and trigger actions at specified times

*Table 26. Triggering and suspend or resume*

| | |
|---|---|
| **ibm.dg.trc.trigger** | Trigger events by tracepoint, group or method entry/exit. |
| **ibm.dg.trc.suspend** | Suspend tracepoints globally (for all threads). |

*Table 26. Triggering and suspend or resume (continued)*

| | |
|---|---|
| **ibm.dg.trc.resume** | Resume tracepoints globally (not really useful, but here for completeness). |
| **ibm.dg.trc.suspendcount** | Initial thread suspend count. |
| **ibm.dg.trc.resumecount** | Initial thread resume count. |

### Properties that specify output files

These properties determine whether trace data is directed to a file. For the first two properties, you must activate tracepoints by using a tracepoint selection property or through the various API that were mentioned earlier. If you specify the `state.output` property, state trace is enabled automatically.

*Table 27. Properties that specify output files*

| | |
|---|---|
| **ibm.dg.trc.output** | Select output file name and options for trace data from tracepoints that were selected through the minimal and maximal properties. |
| **ibm.dg.trc.exception.output** | Select output file name and options for trace data from tracepoints that were selected through the exception property. |
| **ibm.dg.trc.state.output** | Select output file name and options for state trace. |

### MiscellaneousTrace control properties

*Table 28. MiscellaneousTrace control properties*

| | |
|---|---|
| **ibm.dg.trc.properties** | Specify a file containing system properties tor trace. |
| **ibm.dg.trc.buffers** | Modify buffer size and allocation. |
| **ibm.dg.trc.format** | Specify the path for TraceFormat.dat. |

## Detailed property descriptions

The properties are processed in the sequence in which they are described here.

**ibm.dg.trc.properties[=properties_filespec]**

This system property allows you to specify in a file any of the other trace properties, thereby reducing the length of the invocation command line. The format of the file is a flat ASCII/EBCDIC file that contains trace properties; the property names that are in the file can be abbreviated to the name that follows the string `ibm.dg.trc`. If **properties_filespec** is not specified, a default name of **IBMTRACE**.properties is searched for in the java.home directory. Nesting is not supported; that is, the file cannot contain a `properties` property. If any error is found when the file is accessed, DG initialization fails with an explanatory error message and return code. All properties that are in the file are processed in the sequence in which they appear in the file, before the next system property 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 java home:

```
-Dibm.dg.trc.properties
```

Use trace.prop in the current directory:

```
-Dibm.dg.trc.properties=trace.prop
```

Use c:\trc\gc\trace.props:

```
-Dibm.dg.trc.properties=c:\trc\gc\trace.prop
```

Here is an example property file:

```
minimal=all
// maximal=st
maximal=cl
buffers=20k
output=c:\traces\classloader.trc
print=tpid(4002,4005)
```

**ibm.dg.trc.buffers=nnnk | nnnm[,dynamic | nodynamic]**

This property 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** 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 after the first trace entry is lost, and message `UTE018` is issued at JVM termination. For example:

```
UTE115: At least one trace record lost
UTE018: n trace records lost
```

Where *n* is the number of trace records lost.

**Examples:**

Dynamic buffering with 8 KB buffers:

```
-Dibm.dg.trc.buffers=8k
```

or in a properties file:

```
buffers=8k
```

Trace buffers 2 MB per thread:

```
-Dibm.dg.trc.buffers=2m
```

or in a properties file:

```
buffers=2m
```

Trace to only two buffers per thread, each of 128 KB:

```
-Dibm.dg.trc.buffers=128k,nodynamic
```

or in a properties file:

```
buffers=128k,nodynamic
```

**ibm.dg.trc.applids=application_name[,...]**

This system property prepares for trace to be enabled for one or more Java applications that have been instrumented for application trace. The identifier **application_name** must match the name under which the application will register itself. This name can later be used as a component name for tracepoint selection.

**ibm.dg.trc.initialization**

This property allows you to trace JVM initialization. Trace initialization occurs immediately after the XM component initializes the primordial ExecEnv. This action occurs before any classes are loaded or JNI is initialized. As a result, tracing everything for the most simple HelloWorld would produce many MB of output. In most cases, all this information is only noise, so by default, initialization is not traced. If you want to trace initialization, you must specify this system property.

Note that this property allows trace to start before any threads can be created, including the trace write thread. For maximal, minimal, exception, and state tracing, the data has to be buffered until the write thread is started. Ensure therefore that you do not specify the **nodynamic** option on the ibm.dg.trc.buffers system property. Depending on what is being traced, this might cause a large increase in memory usage.

If the failure that is being traced prevents the trace write thread from starting successfully, you can user the **print** form of trace.

**Example:**

Start trace during initialization:

```
-Dibm.dg.trc.initialization
```

**ibm.dg.trc.minimal[=[[!]tracepoint_specification[,...]],**
**ibm.dg.trc.maximal[=[[!]tracepoint_specification[,...]],**
**ibm.dg.trc.count[=[[!]tracepoint_specification[,...]],**
**ibm.dg.trc.print[=[[!]tracepoint_specification[,...]],**
**ibm.dg.trc.iprint[=[[!]tracepoint_specification[,...]],**
**ibm.dg.trc.platform[=[[!]tracepoint_specification[,...]],**
**ibm.dg.trc.exception[=[[!]tracepoint_specification[,...]],**
**ibm.dg.trc.external[=[[!]tracepoint_specification[,...]]**

**Summary**

These properties 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 ibm.dg.trc.output system property.

Tracepoints that are activated with **count** are only counted. The totals are written to dgTrcCounters in the current directory at JVM termination.

Tracepoints that are activated with **print** or **iprint** are routed to stderr.

Tracepoints that are activated with **platform** are routed to the XHPI.

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 ibm.dg.trc.exception.output system property.

**External** trace data is passed to a registered trace listener (see Chapter 35, "Using the Reliability, Availability, and Serviceability interface," on page 355). 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 system properties of the same name.

See "ibm.dg.trc.state.output" on page 335 for information about state trace, which is enabled in a different way, independently of these properties.

**Types of trace**

The **minimal** property records only the timestamp and tracepoint identifier. When the trace is formatted, missing trace data is replaced with ??? in the output file. The **maximal** property specifies that all associated data is traced. If a tracepoint is activated by both system properties, **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** property is specified, it does not affect a later property such as **print**.

The **count** property 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 dgTrcCounters, 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.

The **print** property causes the specified tracepoints to be routed to stderr in real-time. The tracepoints are formatted by TraceFormat.dat, which must be available at runtime. TraceFormat.dat is shipped in sdk/jre/lib and is automatically found by the runtime.

The **platform** property routes trace data to the XHPI in real-time. Currently, none of the XHPI implementations supports this form of tracing, but the infrastructure is in place in the IBM JVM.

The **exception** property 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 property 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" on page 330.

**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** property channels trace data to registered trace listeners in real-time. The JMVRAS interface 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:

- **[!][backtrace[,depth]]component[(type[,...])]** or **[!][backtrace[,depth]]tpid(tracepoint_id[,...])** where

  **!** is a logical not. That is, the tracepoints that are specified immediately following the **!** are turned off.

  **backtrace** indicates that backtrace is activated.

  On some platforms, a C stack-trace can be generated for any tracepoint. By default, 4 levels of call leading to the tracepoint are recorded, but this can be changed by modifying the depth value. This option is of particular use in tracing memory leaks because all memory allocations in the JVM are handled by the dbgmalloc component. By tracing this component, you can identify memory allocations without corresponding frees, and then you can use the stack trace to discover the code where the unmatched memory allocations - the memory leak - are made. The dbgmalloc routines may also be linked into JNI libraries to uncover memory leaks in native code.

  Note that dbgmalloc is meant for IBM use only.

  The backtrace specification applies to all components or tracepoints subsequently selected.

  **depth** is the number of levels of backtrace to record. A depth of 0 indicates that no backtrace is recorded.

**component** is one of:

- ALL.
- The JVM subcomponent (that is, ci, cl, dc, dg, lk, st, xe, xm, hpi, dbgmalloc, java,awt, awt_dnd_datatransfer, Audio, jit, jdwp, mt, fontmanager, net, awt_java2d, awt_print, core, or nio).

  Note that dbgmalloc is meant for IBM use only.
- 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:

  checkedjni, compiledMethods, jitCl, jitCompCtrl, jitEh, jitError, jitInit, jitSt, jni, jvmmi, nativeMethods, st_alloc, st_backtrace, st_calloc, st_compact, st_compact_dump, st_compact_verbose, st_concurrent, st_concurrent_pck, st_concurrent_shadow_heap, st_dump, st_freelist, st_icompact, st_loaAdjustTargetSize, st_mark, st_parallel, st_refs, st_terse, st_trace, st_verbosegc, st_verify, st_verify_heap, staticMethods, threadSR, wrappedjni

**type** is the tracepoint type or **all**. The default is **all**, except for **exception** tracing, where the default is **exception**. The following types are supported:

- Entry
- Exit
- Event
- Exception
- Mem

**tracepoint_id** is the hexadecimal global tracepoint identifier. You can omit leading zeroes. You can specify a range of tracepoints by using a hyphen (dash, minus); for example, tpid(18007,c003-c01f).

**Note:** 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:

```
-Dibm.dg.trc.maximal
```

Eight levels of stack trace to all trace points in the JVM subcomponent dbgmalloc:

```
-Dibm.dg.trc.print=backtrace,8,dbgmalloc
```

Note that dbgmalloc is meant for IBM use only.

All tracepoints except DC and LK:

```
-Dibm.dg.trc.minimal=all,!dc,!lk
```

All entry and exit tracepoints in CL:

```
-Dibm.dg.trc.maximal=cl(entry,exit)
```

All tracepoints in ST except 4000, 4001, 4002, 4003:

```
-Dibm.dg.trc.maximal=st,!tpid(4000,4001,4002,4003)
```

Tracepoints 18005 through 1801f and c003:

```
-Dibm.dg.trc.print=tpid(18005-1801f,c003)
```

All LK tracepoints:

```
-Dibm.dg.trc.count=lk
```

Tracepoints in ST, LK, XE:

```
-Dibm.dg.trc.platform=st,lk,xe
```

All entry and exit tracepoints:

```
-Dibm.dg.trc.external=all(entry,exit)
```

All exception tracepoints:

```
-Dibm.dg.trc.exception
```

All exception tracepoints:

```
-Dibm.dg.trc.exception=all(exception)
```

All exception tracepoints in CL:

```
-Dibm.dg.trc.exception=cl
```

Tracepoints c03e through c113:

```
-Dibm.dg.trc.exception=tpid(c03e-c113)
```

**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.

You can use these keywords either before the tracepoint selection, or as a type modifier. When a keyword is used before a tracepoint selection, that keyword applies to all tracepoint selection criteria that follow it in the system property. For example:

```
-Dibm.dg.trc.maximal=level5,st,lk,cl,level1,all
```

or

```
 -Dibm.dg.trc.maximal=l5,st,lk,cl,l1,all
```

In this example, tracepoints that have a level of 5 or below are enabled for the st, lk, and cl components. Tracepoints that have a level of 1 or below are enabled for all the other components. Note that the level applies only to the current statement, so if multiple trace selection statements appear in a trace properties file, the level is reset to the default for each new statement.

Alternatively, you can specify levels as a type modifier. In this case, the level applies only to the component with which it is associated. The following example is functionally equivalent to the global example that is shown above:

```
-Dibm.dg.trc.maximal=st(level5),lk(level5),cl(level5),all(level1)
```

or

```
-Dibm.dg.trc.maximal=st(l5),lk(l5),cl(l5),all(l1)
```

Level specifications do not apply to explicit tracepoint specifications that use the TPID keyword.

When the not operator is specified, the level is inverted; that is, !st(level5) disables all tracepoints of level 6 or above for the st component. For example:

```
-Dibm.dg.trc.print=all,!lk(l5),!st(l6)
```

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:

```
-Dibm.dg.trc.count=all(l1)
```

Produce maximal trace of all components at level 5 and ST at level 9:

```
-Dibm.dg.trc.maximal=LeVeL5,all,st(L9)
```

Trace all components at level 6, but do not trace dc at all, and do not trace level 0 through 3 entry and exit tracepoints in the XE component:

```
-Dibm.dg.trc.minimal=all(l6),!dc,!xe(level3,entry,exit)
```

**ibm.dg.trc.methods=method_specification[,...]**

This system property identifies which classes and methods are to be prepared to be traced. You can then trace these methods by selecting the MT component 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 system property 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[*]][()]**, where

| | |
|---|---|
| **!** | is a logical not. That is, the class or methods that are specified immediately following the **!** are deselected for method trace. |
| * | is a wildcard that can appear at the beginning, end, or both, of the class and method names. |
| **class** | is the package or class name. 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. |
| **.** | is the delimiter between the class and method. |
| **method** | is the method name. |
| **()** | are left and right parentheses. This specifies that input parameters should be traced where possible. |

**Examples**:

Select all methods for all classes:

```
-Dibm.dg.trc.methods=*
```

All methods that are in java/lang/String. In addition, input parameters and return values should be traced:

```
-Dibm.dg.trc.methods=java/lang/String.*()
```

All methods that contain a "y" in classes that start with com/ibm:

```
-Dibm.dg.trc.methods=com/ibm*.*y*
```

All methods that contain a ″y″ and do not start with an ″n″ in classes that start with com/ibm:

```
-Dibm.dg.trc.methods=com/ibm*.*y*,!n*
```

**ibm.dg.trc.format=TraceFormat_path**

This property overrides the location of TraceFormat.dat when the **ibm.dg.trc.print** system property is specified. Do not specify this property for other forms of tracing, because it uses a large amount of memory with no benefit.

**Example:**

Use c:\formats\122\TraceFormat.dat:

```
-Dibm.dg.trc.format=c:\formats\122
```

**ibm.dg.trc.output=trace_filespec[,nnnm[,generations]]**

This property 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 ÷ 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:

```
-Dibm.dg.trc.output=/u/traces/gc.problem
```

Output goes to trace and will wrap at 2 MB:

```
-Dibm.dg.trc.output= trace,2m
```

Output goes to gc0.trc, gc1.trc, gc2.trc, each 10 MB in size:

```
-Dibm.dg.trc.output=gc#.trc,10m,3
```

Output filename contains today's date in yyyymmdd format (for example, traceout.20031225.trc):

```
-Dibm.dg.trc.output=traceout.%d.trc
```

Output file contains the number of the process (the PID number) that generated it (for example, tracefrompid2112.trc):

```
-Dibm.dg.trc.output=tracefrompid%p.trc
```

Output filename contains the time in hhmmss format (for example, traceout.080312.trc):

```
-Dibm.dg.trc.output=traceout.%t.trc
```

**ibm.dg.trc.exception.output=exception_trace_filespec[,nnnm]**

This property 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 ″yyyymmdd″ 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.

**Examples**:

Trace output goes to /u/traces/exception.trc. No size limit:

```
-Dibm.dg.trc.exception.output=/u/traces/exception.trc
```

Output goes to except and wraps at 2 MB:

```
-Dibm.dg.trc.exception.output=except,2m
```

Output filename contains today's date in yyyymmdd format (for example, traceout.20031225.trc):

```
-Dibm.dg.trc.exception.output=traceout.%d.trc
```

Output file contains the number of the process (the PID number) that generated it (for example, tracefrompid2112.trc):

```
-Dibm.dg.trc.exception.output=tracefrompid%p.trc
```

Output filename contains the time in hhmmss format (for example, traceout.080312.trc):

```
-Dibm.dg.trc.exception.output=traceout.%t.trc
```

**ibm.dg.trc.state.output=state_trace_filespec[,nnnm]**

This property indicates that "state" information should be captured in **state_trace_filespec**. The state trace captures information about the JVM that could be useful later, when the normal trace files or internal buffers have wrapped many times.

Examples of state data might be:

- Interned string values and ids

- Classblock address or name correlation
- Methodblock address or name correlation

A tracepoint is designated as a state-type tracepoint in the TDF at build time. Note that you can also route the tracepoint to another trace destination, such as print, if specified.

State trace differs from other forms of trace in that it is either totally on or off. You cannot control which individual tracepoints are enabled at runtime. By specifying this system property, you turn it on. If **state_trace_filespec** does not already exist, it is created automatically. If it does already exist, it is overwritten. If **nnn** is not specified, the size of the file is not limited. If **nnn** is specified, two files are created. The first file is named state_file_filespec with a 0 (zero) suffix and contains up to nnn MB of state information that is never lost; that is, it never wraps. The second file is named state_file_filespec with a 1 (one) suffix and contains up to nnn MB of state information that wraps; that is, state information might be lost. State trace captures all the startup state information and all the latest state information. The file 0 and 1 filename qualifiers position can optionally be controlled by the inclusion of a # (hash or pound sign) in the filename; the # will be replaced by 0 or 1 respectively. Under normal conditions, nnn should not be specified, but in the case of long-running JVMs, its use might be unavoidable to limit the file size. In this case, some useful state data could be lost.

Optionally, if you want the final trace filename to contain today's date, the PID number that produced the trace, or the time, do one of the following steps as appropriate (see also the examples at the end of this section).

- To include today's date (in "yyyymmdd" format) in the trace filename, specify "%d" as part of the state_trace_filespec.
- To include the pidnumber of the process that is generating the tracefile, specify "%p" as part of the state_trace_filespec.
- To include the time (in 24-hour hhmmss format) in the trace filename, specify "%t" as part of the state_trace_filespec.

**Examples**:

Trace output goes to /u/traces/state; no size limit:

```
-Dibm.dg.trc.state.output=/u/traces/state
```

Output goes to state0 for 4 MB, then state1, wrapping at 4 MB:

```
-Dibm.dg.trc.output= state,4m
```

Output goes to state0.trc for 4 MB, then state1.trc:

```
-Dibm.dg.trc.state.output= state#.trc,4m
```

Output filename contains today's date in yyyymmdd format (for example, traceout.20031225.trc):

```
-Dibm.dg.trc.state.output=traceout.%d.trc
```

Output file contains the number of the process (the PID number) that generated it (for example, tracefrompid2112.trc):

```
-Dibm.dg.trc.state.output=tracefrompid%p.trc
```

Output filename contains the time in hhmmss format (for example, traceout.080312.trc):

```
-Dibm.dg.trc.state.output=traceout.%t.trc
```

**ibm.dg.trc.suspend**

Suspends tracing globally (for all threads and all forms of tracing) but leaves tracepoints activated.

**Example:**

Tracing suspended:

```
-Dibm.dg.trc.suspend
```

**ibm.dg.trc.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 start-up option):

```
-Dibm.dg.trc.resume
```

**ibm.dg.trc.highuse**

Some routines in the JVM are very frequently entered. Any tracepoints in them would adversely affect performance. So, under normal conditions, it is not possible to trace these routines. To overcome this, you can, at runtime, use the **highuse** property to select duplicate routines that have tracepoints incorporated. You can set this for all, or single, components. Activate the trace points in the standard way. The components that have **highuse** trace incorporated are: cl, dc, lk, st, jni, and hpi.

**Examples:**

```
-Dibm.dg.trc.highuse=all
-Dibm.dg.trc.highuse=st,cl
```

**ibm.dg.trc.suspendcount=count**

This system property is for use with the **ibm.dg.trc.trigger** property (see "ibm.dg.trc.trigger" on page 338).

This **ibm.dg.trc.suspendcount=count** system property 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** suspendthis actions 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:

```
-Dibm.dg.trc.suspendcount=3
```

**ibm.dg.trc.resumecount=count**

This system property is for use with the **ibm.dg.trc.trigger** property (see "ibm.dg.trc.trigger" on page 338).

This **ibm.dg.trc.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** resumethis actions 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:

```
-Dibm.dg.trc.resumecount=3
```

**ibm.dg.trc.trigger=clause[,clause][,clause]...**
This property 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 property does not control what is traced. It controls only whether what has been selected by the other trace properties is produced as normal or is blocked.

Each clause of the trigger property 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(methodspec,[entryAction],[exitAction][,delayCount][,matchcount])**
On entering a method that matches **methodspec**, perform the specified **entryAction**. On leaving it, perform the specified **exitAction**. If you specify a **delayCount**, the actions are performed only after a matching **methodspec** has been entered that many times. If you specify a **matchCount**, **entryAction** and **exitAction** 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 **tpid** (tracepoint id) or a tpid that falls inside the specified **tpidRange**, perform the specified action. If you specify a **delayCount**, the action is performed only after the JVM finds such an active tpid that many times. If you specify a **matchCount**, **action** will be performed at most that many times.

**threshold(thresholdType,thresholdValue,[actionOn],**
**[actionOff][,delayCount][,matchcount])**
Triggering occurs depending on the value of the t**hresholdType**. For v1.4.2, the only supported **thersholdType** is *"heapfree"*. When the percentage of the heap that is free falls to **thresholdValue** or below, after a GC cycle, **actionOn** is performed; when the percentage of the heap that is free rises again to above the **thresholdValue**, **actionOff** is performed. If you specify a **delayCount**, the actions are performed only after the time it would take for **actionOn** to perform that many times. If you specify a **matchCount**, **actionOn** and **actionOff** 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 ibm.dg.trc.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.

**resumethis**
Decrement the suspend count for this thread. If the suspend-count is zero or below, tracing for this thread is resumed.

**coredump (or sysdymp)**
Produce a coredump.

**javadump**
Produce a javadump or javacore.

**heapdump**
Produce a heap dump (see Chapter 26, "Using Heapdump," on page 245).

**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.hhmmssth.ppppp.trc`, where `nnnn` is the sequence number of the snap file since JVM startup, `yyyymmdd` is the date, `hhmmssth` is the time, and `ppppp` is the process id in decimal with leading 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:

  ```
  -Dibm.dg.resumecount=1
  -Dibm.dg.trc.trigger=method(java/lang/String.*,resumethis,suspendthis)
  ```

- Resume all tracing when any thread enters a method in any class that starts with "error":

  ```
  -Dibm.dg.trc.trigger=method(*.error*,resume)
  ```

- When you reach the 1000th and 1001st tracepoint from the ″jvmmi″ trace group, produce a core dump.

  **Note:** Without matchcount there would be a risk of filling your disk with coredump files.

  ```
  -Dibm.dg.trc.trigger=group(jvmmi,coredump,1000,2)
  ```

- Trace (all threads) while my application is active only; that is, not startup or shutdown. (The application name is "HelloWorld"):

  ```
  -Dibm.dg.trc.suspend
  -Dibm.dg.trc.trigger=method(HelloWorld.main,resume,suspend)
  ```

- The first time (only) that there is less than or equal to 25% heap free, snap the trace, when the percentage of heap free climbs back above 25%, take a coredump:

  ```
  -Dibm.dg.trc.trigger=threshold(heapfree,25,snap,sysdump,1)
  ```

# 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.
- `entries:comp[,...]` specifies that only trace entries for component comp are to be formatted.
- `thread:threadid,...` specifies that only entries for threadid are to be formatted (threadid is specified as 0xnnnnnnnn).
- `indent` specifies that the trace data is to be indented on entry type tracepoints and outdented on exit type tracepoints. This might produce undesirable results on selective traces where, for example, exits from a function are not traced, but entries are.
- `symbolic` specifies that the symbolic name of the tracepoint is embedded in the trace output. This is useful where the descriptive text for a particular tracepoint does not make clear what is being traced.

Examples of formatting binary trace file `trace1`:

- Produce a summary of the trace file:

  ```
  java com.ibm.jvm.format.TraceFormat trace1 -summary
  ```

- Format trace1 using the formatting templates (TraceFormat.dat) that are in d:\formats:

  ```
  java -Dibm.dg.trc.format=d:\formats com.ibm.jvm.format.TraceFormat trace1
  ```

- Format trace1 indenting for entry tracepoints and outdenting for exits:

  ```
  java com.ibm.jvm.format.TraceFormat trace1 -indent
  ```

- Format only the trace information in trace1 that originated from the XE component:

  ```
  java com.ibm.jvm.format.TraceFormat trace1 -entries:xe
  ```

- Format only the trace information in trace1 that originated from the thread that has an execenv address of 0x7ffee00:

  ```
  java com.ibm.jvm.format.TraceFormat trace1 -thread:0x7ffee00 -indent
  ```

# Trace properties

The use of properties files for controlling trace not only saves typing, but, over time, causes a library of these files to be created, with each file tailored to solving problems in a particular area. Especially useful is the ability to remove unwanted tracepoints by using the **!TPID(xxxxxx)** parameter.

### What to trace

JVM trace can produce large amounts of data in a very short time. Before running trace, carefully think about what information you require to solve the problem. In many cases, only the trace information that is produced shortly before the problem needs to be captured, so using the wrap option on the output trace file should be considered. In many cases, it is enough to 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 automatically snapped to a file that is in the current directory. The file is called: Snap*nnnn.yyyymmdd.hhmmssth.process*.trc.

Also carefully think about which components need to be traced and what level of tracing is required. For example, if a suspected garbage collection problem is being traced, it might be enough to trace all components at level 1 or 3, and ST 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

Each tracepoint has a unique 3-byte identifier (6 hex digits). This identifier relates the tracepoint in the code to its entry in the format file (jre/lib/TraceFormat.dat). You can use the identifier to select individual tracepoints at runtime by using the TPID keyword. The tracepoint ID can be looked up in the format file, which has the following format:

The first line is an internal version number.

Following the version number is a component name, followed by a line for each tracepoint defined in that component, the format of which for this JVM is: **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 used to format the entry.

Note that this is subject to change without notice, but the version number will be different.

## Using trace to debug memory leaks

The JVM comes with a library dbgmalloc, which wraps the calls to memory routines such as malloc() and free(). Used with the JVM trace facility, all calls to memory routines from libraries linked with dbgmalloc can be logged. By post-processing the trace output, memory allocations that do not have a corresponding free() can be identified. The **backtrace** trace option is also very useful to identify the caller of the relevant memory allocations.

Note that dbgmalloc is meant for IBM use only.

Dbgmalloc is linked in with the JVM and most of the native libraries and can be linked in to customer libraries. The platform-specific sections for debugging memory leaks tell you how to do this. The method varies according to the operating system. See:

- "Debugging memory leaks" on page 103 for AIX
- "Debugging memory leaks" on page 138 for Linux

- "Debugging memory leaks" on page 161 for Windows
- "Debugging memory leaks" on page 182 for z/OS

Note that when the JVM starts up, a lot of memory is allocated for its own purposes. Much of this memory will never be deallocated, but this does not constitute a leak. Also, the JIT allocates memory whenever it decides that a method should be compiled. Again, this does not constitute a leak.

## Enabling memory tracing

Run the jvm with **-Dibm.dg.trc.print=dbgmalloc** and direct the standard error to a large file. When this file is analyzed to find a memory leak, the start of the file may be discarded because the file will largely consist of memory required for JVM startup.

Another approach is to collect trace in memory; a fixed size buffer is used and older data is overwritten. For example, the settings **-Dibm.dg.trc.minimal=dbgmalloc -Dibm.dg.trc.buffers=2M** will use 2 MB of buffer for each thread. To write out the memory buffers, trace must be snapped by sending a signal to the Java process.

## Enabling backtrace

Backtrace is enabled by preceding the dbgmalloc specification with *backtrace*; for example **-Dibm.dg.trc.print=backtrace,dbgmalloc**. If the default number of routines traced is insufficient to identify the caller, the depth of the stack trace may be increased. For example, to set the depth to 8, type: **-Dibm.dg.trc.print=backtrace,8,dbgmalloc**.

## Linking with dbgmalloc

dbgmalloc must be linked in to JNI libraries so that its versions of the memory routines are called before the system ones. No change is required to the JNI code. The linker options to do this differ by operating system.

# Chapter 34. Using the JVM monitoring interface (JVMMI)

The JVM Monitoring Interface (JVMMI) is a lightweight interface for monitoring the behavior of the JVM. It is similar to the JVM Profiling Interface (JVMPI). However, JVMPI is a ″heavyweight″ diagnostic tool that is unlikely to be used in a production environment. JVMMI provides the basic monitoring of JVMPI without its performance overhead; thus, you can leave it running throughout the lifecycle of a JVM.

JVMMI allows you to monitor JVM activity. For example, you might want to know:

- When garbage collection starts or finishes
- When the heap is expanded
- When a thread is created, started, or ended
- When the JVM initialization is complete
- When the JVM is ending

Events are coded at these points and others. (See “Events produced by JVMMI” on page 348 for a full list.) When an enabled event is reached, a registered function in a user-provided agent DLL is called back.

Because JVMMI must be very lightweight, callbacks are run on the thread that caused the event. Therefore, the thread does not continue its normal activity until the callback function returns. For this reason, your callback functions should be as brief as possible.

Note that you need some programming skills to use this interface. To use the JVMMI, you must be able to build a native library, add the code for JVMMI callbacks (described below), and to interface the code to the JVM via the JNI. This book provides the callback code and gives simple examples of how to build a JVMMI agent (see “Building the agent” on page 346). It does not provide the other programming information.

This chapter describes how to use the JVMMI in:

- “Using JVMMI for problem determination”
- “Preparing to use JVMMI” on page 344
- “API calls provided by JVMMI” on page 347
- “Events produced by JVMMI” on page 348
- “Enumerations supported by JVMMI” on page 351

## Using JVMMI for problem determination

Although the interface is primarily for monitoring, it can also be useful in problem determination. Here are some situations and some possible uses for JVMMI:

**The JVM appears to be running more and more slowly**
Possibly the heap is filling up. Try tracking the events for heap size changes, garbage collection start and finish, area alloc, heap low and heap full. If the JVM continues to allocate objects (area alloc events), heap full events are occurring regularly, and the time between garbage collections is constantly decreasing, there could be a problem with the design of the application. The

application might be generating too much data or is keeping references to out-of-date data that could otherwise have been garbage collected.

**A server fails to respond to a client connection**
If the server starts a thread to talk to each client, check the threads (creation, starting, stopping, and thread enumeration) to see how many threads are running at a given time. If you know what the session threads are called, check whether the session's thread is being created and whether it is being started.

**A Java program does not start (appears to hang)**
Enable the JVM Init Done event to see whether the problem is during initialization or during the running of your application. Check class load events to see if your classes are being loaded.

# Preparing to use JVMMI

The following sections explain how to extract monitoring information from the JVM.

## Writing an agent

To use the JVMMI interface, you must first write an agent DLL. The following code is a complete JVMMI agent. It enables one event (**JVMMI_EVENT_JVM_INIT_DONE**) and prints a message to stdout when this event is received. The code demonstrates how to obtain the JVMMI interface and enable an event. (Furthermore, as you can see, writing a JVMMI agent does not have to be a big project).

```
#include <stdio.h>
#include <jni.h>
#include <jvmmi.h>

/* This is a piece of user data.  I'm not really using it in this agent. */
char ud[] = "Not really used in this example";


/*
 * This routine will be called back when a JVMMI_EVENT_JVM_INIT_DONE event
 * is received.
*/
int JNICALL JVMInitDone(JNIEnv *env, JVMMI_Event *evt, void *userData, int tid)
{
    printf("----- JVM_INIT_DONE -----\n");
    return JVMMI_OK;
}


/*
 * This routine will be called immediately the agent DLL is loaded by java
 * with the -Xrun option.  It registers the above function as a callback for
 * the JVM init done event.
 */
JNIEXPORT jint JNICALL JVM_OnLoad(JavaVM *vm, char *options, void *reserved)
{
    JNIEnv            *JNI_env;
    JVMMI_Interface   *JVMMI_env;
    jint JVMMIversion = JVMMI_VERSION_1;
    jint JNIversion   = JNI_VERSION_1_2;


    /* Obtain the JNI interface */
    if (JNI_OK != (*vm)->GetEnv(vm, (void **)&JNI_env, JNIversion)) {
        printf(stderr,"Failed to get JNI Env\n");
        return JVMMI_ERR
```

```
    }

    /* Obtain the JVMMI interface */
    if (JNI_OK != (*vm)->GetEnv(vm, (void **)&JVMMI_env, JVMMIversion)) {
        printf(stderr,"Failed to get JVMMI Env\n");
        return JVMMI_ERR
    }

        /* Enable the JVM Init Done event */
    if (JVMMI_OK != JVMMI_env->EnableEvent(JNI_env,
                                           JVMInitDone,
                                           ud,
                                           JVMMI_EVENT_JVM_INIT_DONE)) {
        printf(stderr,"EnableEvent JVMMI_EVENT_JVM_INIT_DONE failed");
        return JVMMI_ERR
    }

    return JVMMI_OK;
}
```

To run this program, type: `java -Xrunmyagent HelloWorld`. If it is working correctly you will receive the following output:

```
----- JVM_INIT_DONE -----
Hello World.
```

## Using Detail information in a JVMMI agent

You will find a definition of the Detail information that accompanies the events in jvmmi.h (in the sdk/include/ directory). Here is an example of how you might use it.

First add the following code into the JVM_OnLoad function to enable the Thread Created event (JVMMI_EVENT_THREAD_CREATED) in the simple agent above:

```
    /* Enable the Thread Created event */
    if (JVMMI_OK != JVMMI_env->EnableEvent(JNI_env,
                                           ThreadCreated,
                                           ud,
                                           JVMMI_EVENT_THREAD_CREATED)) {
        printf(stderr,"EnableEvent JVMMI_EVENT_THREAD_CREATED failed");
        return JVMMI_ERR
    }
```

Then add the new function below to the simple agent:

```
int JNICALL ThreadCreated(JNIEnv *env, JVMMI_Event *evt, void *userData,int tid)
{
    printf("\nTHREAD CREATED event received\n");

    printf("name of creating thread:        %s\n",
           evt->detail.thread_info.name);

    printf("id of creating thread:          0x%p\n",
           evt->detail.thread_info.id);

    printf("id of creating thread's parent: 0x%p\n",
           evt->detail.thread_info.parent_id);

    printf("id of new (child) thread:       0x%p\n",
           evt->detail.thread_info.child_id);

    return JVMMI_OK;
}
```

Now, every time that a thread is created you will obtain information about it. The output from the event above might look like this:

```
THREAD CREATED event received
name of creating thread:        "main"
id of creating thread:          0x00235190
id of creating thread's parent: 0x00000000
id of new (child) thread:       0x3224AB68
```

The parent of the main thread always has an id of 0. In a complex program, the thread id's can be used to build a thread hierarchy. However, in a simple program like 'HelloWorld', everything (except "main") is started by "main".

## Using user data in a JVMMI agent

When you enable an event you must provide a (void *) pointer to a piece of user data. In the example above, there is only a placeholder, but it gives the agent some flexibility. For example, you might want to modify the ThreadCreated code above to store the id's of the threads you see being created.

## Using Detail information on EBCDIC platforms

On z/OS, the Detail information that is returned with events contains a (char *) pointer. This is a pointer to an ASCII string. To use the information on z/OS, copy the data to your own storage and perform an ASCII-to-EBCDIC conversion on it. You can write something like this:

```
#include <unistd.h>
char buffer[128]="";
...
strcpy(buffer, evt->detail.thread_info.name);
__atoe(buffer);
```

## Obtaining the JVMMI interface

Before you can make any JVMMI calls, you must obtain the JVMMI interface (this is very similar to obtaining the JNI interface). Use the VM **GetEnv** function, for example:

```
JNIEnv *JNI_env;
      .....
 rc = (*vm)->GetEnv(vm, (void **)&JNI_env, JVMMI_VERSION_1);
```

## Specifying the agent name

When you start the JVM, you specify the name of the agent DLL that uses the **-Xrun** option. For example:

```
Java -Xrunmyagent HelloWorld
```

## Inside the agent

When the JVM starts, it loads your agent and calls its JVM_OnLoad function. This acquires the JVMMI interface as above. The function then enables some, or all, of the provided events, supplying a callback function for each of the ones it enables.

## Building the agent

### Windows

Before you can build a JVMMI 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 JVMMI agent, enter the command:

```
cl /MD /Femyagent.dll myagent.c  /link /DLL
```

### Linux

To build a JVMMI 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.

### AIX PPC32

To build a JVMMI 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
xlC_r -bM:UR -qcpluscmt $INCLUDE_DIRS $JVM_LIB -ljvm -o libmyagent.so myagent.c
```

### AIX PPC64

To build a JVMMI 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 myagent.c
```

### z/OS

To build a JVMMI 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.

## API calls provided by JVMMI

The JVMMI interface contains three functions:

### EnableEvent

```
int JNICALL EnableEvent(JNIEnv *env,
                jvmmi_callback_t func,
                void *userData,
                int eventId);
```

**Description**

Enables an event

**Parameters**

- A pointer to the JNI environment.
- The function you want to be called whenever the event occurs.
- A pointer to a piece of user data.

- The number of the event in which you are interested (for example, JVMMI_EVENT_CLASS_LOAD).

**Returns**

If the callback is successfully registered, the function returns JVMMI_OK. If the registration is unsuccessful, the function returns JVMMI_ERR.

## DisableEvent

```
int JNICALL DisableEvent(JNIEnv *env,
                 jvmmi_callback_t func,
                 void *userData,
                 int eventId);
```

**Description**

Disables an event.

**Parameters**

- A pointer to the JNI environment.
- The function you no longer want to be called whenever the event occurs.
- The pointer to the user data you supplied when you enabled the event.
- The number of the event in which you are no longer interested.

**Returns**

If the callback is successfully deregistered, the function returns JVMMI_OK. If the deregistration is unsuccessful, the function returns JVMMI_ERR.

## EnumerateOver

```
int JNICALL EnumerateOver(JNIEnv *env,
                      int itemType,
                      int limit,
                      jvmmi_callback_t func,
                      void *userData);
```

**Description**

Finds all items of the specified type (itemType) and calls the supplied callback function (func()) for each one.

**Parameters**

- A pointer to the JNI environment.
- The type of object you want to enumerate (for example, JVMMI_LIST_MONITOR).
- The maximum number of times you want to be called back.
- The function you want to be called for each item found.
- A pointer to a piece of user data.

**Returns**

If the enumeration is performed successfully, the function returns JVMMI_OK. If the enumeration is unsuccessful, the function returns JVMMI_ERR.

# Events produced by JVMMI

A typical call to enable an event (in this case, garbage collection start) might look like this:

```
JVMMI_env->EnableEvent(env,                  // JNI environment
                      agentGCStart,          // call agentGCStart func when hit
                      userData,              // supplying this user data
                      JVMMI_EVENT_GC_START); // which event?
```

The JVMMI interface is self-describing. To find out which events your build of the JVM supports, use the enumeration JVMMI_LIST_EVENTS. This enumeration returns information about the events, including detailed information returned with them and the values of the JVMMI_EVENT_* constants. In this way, if further events are added in a service PTF, or in future development releases, you will be able to use them.

The events are of the following types:

- Thread-related events
- Class-related events
- Heap and garbage collection events
- Miscellaneous events

## Thread-related events

**JVMMI_EVENT_THREAD_CREATION_REQUESTED**
This special event is produced by the JVM before it attempts to create a new thread. If enabled, your callback function must return TRUE if you want to allow the thread to be created. If it returns anything else, the thread is not created and a Java SecurityException is thrown.

Detailed information identifies the id and name (for example, ″main″) of the creating thread, and the id of its parent.

**JVMMI_EVENT_THREAD_CREATED**
This event is produced when a thread is created. It is produced by the creating thread after it has created the child thread but before is starts it.

Detailed information identifies the id and name of the creating thread, the id of its parent, and the id of the new (child) thread.

**JVMMI_EVENT_THREAD_START**
This event is produced by a new thread, confirming that it has begun.

Detailed information identifies the id and name of the starting thread and the id of its parent.

**JVMMI_EVENT_THREAD_STOP**
This event is produced by a thread just before it terminates.

Detailed information identifies the id and name of the stopping thread and the id of its parent.

## Class-related events

**JVMMI_EVENT_CLASS_LOAD**
This event informs you that a class has been loaded into the heap.

Detailed information identifies the class name, the source file name, the number of interfaces, the number of methods, and the number of fields.

**JVMMI_EVENT_CLASS_UNLOAD**
This event informs you that a class has been removed from the heap.

Detailed information identifies the class name, the source file name, the number of interfaces, the number of methods, and the number of fields.

## Heap and garbage collection events

**JVMMI_EVENT_GC_MARK_START**
This event informs you that the storage manager has begun a parallel mark of the heap preceding a garbage collection.

Detailed information identifies the free space, the expected free space (and the kickoff threshold).

**JVMMI_EVENT_GC_MARK_STACK_OVERFLOW (new for 1.4.1)**
This event tells you that a mark stack overflow was detected during concurrent mark.

Detailed information contains a counter for the number of mark stack overflows that have occurred.

**JVMMI_EVENT_GC_START**
This event informs you that a garbage collection has commenced.

Detailed information identifies the heap that is being collected, the size of the heap before garbage collection, the size of the heap after garbage collection, the amount of free space in the heap, the number of the garbage collection, the number of the allocation failure, the percentage time in garbage collection (over the previous 10 garbage collections), and the largest available free chunk of memory.

**JVMMI_EVENT_GC_COMPACT (new for 1.4.1)**
This event tells you that the garbage collector has compacted the heap.

Detailed information identifies the heap that is being collected, the size of the heap, and the amount of free space in the heap.

**JVMMI_EVENT_GC_FINISH**
This event informs you that the garbage collection has ended.

Detailed information identifies the heap that is being collected, the size of the heap, and the amount of free space in the heap.

**JVMMI_EVENT_AREA_ALLOC**
This event is produced periodically, each time 1 MB of storage has been used for object allocation.

Detailed information identifies how much memory has been allocated since the last area alloc event (or, for the first such event, since the event has been enabled).

**JVMMI_EVENT_HEAP_LOW**
This event informs you that one of the heaps is running low on available memory. An object allocation request to this heap resulted in extensive garbage collection to fit in the new object.

Detailed information identifies the heap that is running low on memory, the size of the heap, and the amount of free space in the heap.

**JVMMI_EVENT_HEAP_SIZE**
This event informs you that the size of one of the heaps has been modified.

Detailed information identifies the heap that has altered in size, the size of the heap before this alteration, the size of the heap after it, and the amount of free space in the heap.

**JVMMI_EVENT_HEAP_FULL**
This event informs you that one of the heaps is full. An object allocation request to this heap has just failed.

Detailed information identifies the heap that is full, the size of the heap, and the amount of free space in the heap.

**JVMMI_EVENT_OUT_OF_MEMORY**
This event is produced whenever a Java out-of-memory error is thrown.

Detailed information identifies the amount of memory allocated, the maximum size of the heap, the amount of free memory remaining on the freelist, the number of garbage collection cycles performed, and the detail string accompanying the exception.

**JVMMI_EVENT_GC_OBJECT_ENUMERATE (new for 1.4.1)**
This event follows the enumeration of objects.

Detailed information tells you whether the type of enumeration that was performed (standard or accurate) and the number of objects that were found in each of the regular (main) heap, the transient heap, the system heap, the ACS heap, and the transient local heaps.

**JVMMI_EVENT_OBJECT_REFERENCES**
If enabled, this event follows each JVMMI_LIST_OBJECT event. It contains the references from this object to other objects for use in heap analysis. If the object has many references, this event might occur multiple times. Detailed information includes the references themselves, whether further JVMMI_EVENT_OBJECT_REFERENCES will follow, and the summary information about the object itself.

### Miscellaneous events

**JVMMI_EVENT_JVM_INIT_DONE**
This event is produced when the JVM has finished its initialization and is ready to start processing a user program.

No detailed information accompanies this event.

**JVMMI_EVENT_JVM_SHUTDOWN**
This event is produced when the JVM is shutting down, at the last point it can guarantee to get an event out before shutdown. Because of multithreading issues, it is usually not the last event produced.

No detailed information accompanies this event.

**JVMMI_SERVICE_EVENT_HEAPDUMP**
This event is produced when the JVM gets a signal, and the JAVA_DUMP_OPTIONS environment variable requires a Heapdump to be generated for this signal.

No detailed information accompanies this event.

To use this event, you must include this `#define` in your agent:

```
#define JVMMI_SERVICE_EVENT_HEAPDUMP 1001
```

## Enumerations supported by JVMMI

A typical call to perform an enumeration (in this case of threads) might look something like this:

```
JVMMI_env->EnumerateOver(env,              // JNI environment
                         JVMMI_LIST_THREAD, // type of enumeration, enum threads
                         30,               // callback a maximum of 30 times
                         agentThreads,     // call agentThread func each time
                         userData);        // supplying this user data
```

The JVMMI interface is self-describing. To find out which enumerations your build of the JVM supports, use the enumeration JVMMI_LIST_DEFINITION. This returns information about the enumerations, including the values of the JVMMI_LIST_* constants. In this way, if further enumerations are added in a service PTF, or in future development releases, you will be able to use them.

This is the full list of enumerations that are currently supported:

**JVMMI_LIST_COMPONENT**
The supplied routine is called back once for each JVM component (for example, ST, XE, CI).

**JVMMI_LIST_DEFINITION**
The supplied routine is called back once for each type of enumeration supported by JVMMI.

**JVMMI_LIST_EVENT**
The supplied routine is called back once for each event supported by JVMMI.

**JVMMI_LIST_MONITOR**
The supplied routine is called back once for each monitor found. Note that this call must scan all the objects on the heap and so might take some time to complete.

**JVMMI_LIST_OBJECT**
The supplied routine is called back once for each object found on the heaps. This call must scan the heap and so might take some time to complete. The simplest "HelloWorld" program generates several thousand objects, so you must expect a lot of calls.

**JVMMI_LIST_THREAD**
The supplied routine is called back once for each current thread.

**JVMMI_LIST_SYSLOCK**
The supplied routine is called back once for each JVM internal system lock.

## Sample JVMMI Heapdump agent

The example program shown below is a basic Heapdump agent. When the heap becomes full, or when a signal occurs for which **JAVA_DUMP_OPTS** specifies "HEAPDUMP", this agent is called. The agent creates and writes to a file called heapdump.txt that contains a list of the objects on the heap, along with their references to other objects.

You can post-process this file using HeapRoots (for more information, see "Using the HeapRoots post-processor to process Heapdumps" on page 249).

For instructions on how to build JVMMI agents, see "Writing an agent" on page 344.

```
#include <stdio.h>
#include <jni.h>
#include <jvmmi.h>
#define          FALSE 0
#define          TRUE  1
FILE             *hdfile;
JavaVM           *vm = NULL;
JNIEnv           *JNI_env;
JVMMI_Interface  *JVMMI_env;
char             ud[] = "Not_Used";
int              heapdumpObjectLimit=1000000;
int              new_object;
```

```
int JNICALL callbacks(JNIEnv *env, JVMMI_Event *evt, void *userData, int tid) {
    switch(evt->type) {

    case (JVMMI_EVENT_HEAPDUMP) :
        fprintf(stderr, "JVMMI agent - HEAPDUMP started\n");
        hdfile = fopen("heapdump.txt", "w");

        /*
         * Enumerate over all the objects on the heap
         */
        JVMMI_env->EnumerateOver(env,
                                 JVMMI_LIST_OBJECT,
                                 heapdumpObjectLimit,
                                 callbacks,
                                 userData);

        fprintf(stderr, "JVMMI agent - HEAPDUMP complete\n");
        fclose(hdfile);
        break;

    case (JVMMI_LIST_OBJECT) :
        /*
         * called once per object.
         * Set a flag to say this is a new object.
         */
        new_object=TRUE;
        break;

    case (JVMMI_EVENT_OBJECT_REFERENCES) :
        /*
         * Write out the object description only on the first time
         * here for each object.  This contains address and type.
         */
        if(new_object) {
            fprintf(hdfile, "%s",
                    evt->detail.reference_info.object_description);
            new_object = FALSE;
        }

        /*
         * print out the references from this object to other objects
         * if more_follow is set in the detail information, then
         * another event will follow, containing more references.
         */
        fprintf(hdfile, "%s", evt->detail.reference_info.refs);

        /*
         * if we're done throw a newline
         */
        if(!evt->detail.reference_info.more_follow)
            fprintf(hdfile, "\n");

        break;
    }
    return JVMMI_OK;
}

/* This is run when the DLL is loaded */
JNIEXPORT jint JNICALL JVM_OnLoad(JavaVM *vm, char *options, void *reserved) {

    jint    JVMMIversion = JVMMI_VERSION_1;
    jint    JNIversion   = JNI_VERSION_1_2;

    /*
     * Get the JNI and JVMMI interfaces
     */
```

```
    (*vm)->GetEnv(vm, (void **) &JNI_env, JNIversion);
    (*vm)->GetEnv(vm, (void **) &JVMMI_env, JVMMIversion);

    /*
     * Enable events for heapdump and object references
     */
    JVMMI_env->EnableEvent(JNI_env, callbacks, ud,
                           JVMMI_EVENT_HEAPDUMP);
    JVMMI_env->EnableEvent(JNI_env, callbacks, ud,
                           JVMMI_EVENT_OBJECT_REFERENCES);

    return JVMMI_OK;
}
```

# Chapter 35. Using the Reliability, Availability, and Serviceability interface

The JVM Reliability, Availability, and Serviceability Interface (JVMRI) allows an agent or plug-in to access reliability, availability, and serviceability (RAS) functions by using a facade (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

Note that you need some programming skills to use this interface. 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 via the JNI. This book provides the callback code but does not provide the other programming information.

This chapter describes the JVMRI in:



**Note:** The JVMRI was originally called JVMRAS. You might find references to JVMRAS in the code.

## Preparing to use JVMRI

Before you can use the JVMRI Trace API, you must enable at least one of the **–Dibm.dg.trc.<nnnn>** options.

### 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 "jvmras.h"

DgRasInterface *jvmri_intf = NULL;

JNIEXPORT jint JNICALL
```

```
JVM_OnLoad(JavaVM *vm, char *options, void *reserved)
{
    int    rc;
    JNIEnv *env;

    /*
     * Get a pointer to the JNIEnv
     */

    rc = (*vm)->GetEnv(vm, (void **)&env, JNI_VERSION_1_2);
    if (rc != JNI_OK) {
        fprintf(stderr, "RASplugin001 Return code %d obtaining JNIEnv\n", rc);
        fflush(stderr);
        return JNI_ERR;
    }

    /*
     * Get a pointer to the JVMRI function table
     */

    rc = (*vm)->GetEnv(vm, (void **)&jvmri_intf, JVMRAS_VERSION_1_3);
    if (rc != JNI_OK) {
        fprintf(stderr, "RASplugin002 Return code %d obtaining DgRasInterface\n", rc);
        fflush(stderr);
        return JNI_ERR;
    }

    /*
     * Now a pointer to the function table has been obtained we can make calls to any
     * of the functions in that table.
     */

     ....................................................

    return rc;
}
```

## Registering a trace listener

Before you start using the trace listener, you must set the **-Dibm.dg.trc.external** option to inform the object of the tracepoints for which it should listen.

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 via 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 'st', use the following command when launching the JVM and your agent:

```
java -Xrunjvmri:maximal=st  -Dibm.dg.trc.external=ci App.class
```

**Note:** Trace must be enabled before the agent can be used. To do this, specify the trace option on the command line: `-Dibm.dg.trc.external=ci`.

## Launching the Agent

To launch the agent when the JVM starts up, use the **–Xrun** command line option. For example if your agent is called jvmri, specify `-Xrunjvmri: <options>` on the command line.

## Building the agent

The procedure for building a JVMRI agent is the same as the procedure for JVMMI. For information on how to build an agent, see the section JVMMI — Building the agent in Chapter 34, "Using the JVM monitoring interface (JVMMI)," on page 343.

## Plug-in design

The plug-in must reference the header files jni.h and jvmras.h, which are shipped with the SDK and are in the sdk\include subdirectory. To launch the plug-in, 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 plug-in can then call the functions that have been initialized, by using the JVMRI facade.

## JVMRI functions

At startup, the JVM initializes RI functions. You access these functions with the JNI GetEnv routine to obtain an interface (facade) 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 listed in the sequence in which they appear in the facade.

### 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**

- Pointer to a JNIEnv.
- Function pointer to trace function to register.

**Returns**
JNI Return code JNI_OK or JNI_ENOMEM.

### 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**

- Pointer to a JNIEnv.
- Function pointer to a previously-registered trace function.

**Returns**
JNI Return code JNI_OK or JNI_EINVAL.

### TraceSet

```
int TraceSet(JNIEnv *env, const char *cmd)
```

**Description**
Sets the trace configuration options.

**Parameters**
- 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**
- 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**
- Pointer to a JNIEnv. Must be valid, if MULTI_JVM; otherwise, it can be NULL.

**Returns**
Nothing.

## TraceResume

```
void TraceResume(JNIEnv *env)
```

**Description**
Resumes tracing.

**Parameters**
- Pointer to a JNIEnv. Must be valid, if MULTI_JVM; otherwise, it can be NULL.

**Returns**
Nothing.

## 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**

- Pointer to a JNIEnv.
- Function pointer to trace function to register.

**Returns**
JNI return codes JNI_OK and JNI_ENOMEM.

## 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**

- Pointer to a JNIEnv.
- Function pointer to trace function to register.

**Returns**
JNI return codes JNI_OK and JNI_EINVAL.

## NotifySignal

```
void NotifySignal(JNIEnv *env, int signal)
```

**Description**
Raises a signal in the JVM.

**Parameters**

- Pointer to a JNIEnv. This parameter is reserved for future use.
- Signal number to raise.

**Returns**
Nothing.

## 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 364.

**Parameters**

- 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.

## 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**
- 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 364.)

**Returns**
JNI Return codes `JNI_OK` or `JNI_EINVAL`.

## CreateThread

```
int CreateThread( JNIEnv *env, void JNICALL (*startFunc)(void*),
      void *args, int GCSuspend)
```

**Description**
Creates a thread. If GCSuspend is not 0, the thread is not suspended when garbage collection is performed. A thread can be created only after the JVM has been initialized. Calls to `CreateThread` can take place before initialization, but the threads are created only after initialization by a callback function.

**Parameters**
- Pointer to a JNIEnv.
- Pointer to start function for the new thread.
- Pointer to argument that is to be passed to start function.
- Flag that indicates whether or not thread will be suspended when garbage collection occurs.

**Returns**
JNI Return code `JNI_OK` if thread creation is successful; otherwise, `JNI_ERR`.

## GenerateJavacore

```
int GenerateJavacore( JNIEnv *env )
```

**Description**
Generates a Javacore file.

**Parameters**
- Pointer to a JNIEnv.

**Returns**
JNI Return code `JNI_OK` if running dump is successful; otherwise, `JNI_ERR`.

## RunDumpRoutine

```
int RunDumpRoutine( JNIEnv *env, int componentID, int level, void (*printrtn)
 (void *env, const char *tagName, const char *fmt, ...) )
```

**Description**
Runs one of the individual registered dump routines. Output is sent to the supplied print routine.

**Parameters**
- Pointer to a JNIEnv.
- Id of component to dump.

- Detail level of dump.
- Print routine to which dump output is directed.

**Returns**
JNI Return code JNI_OK if running dump is successful; otherwise, JNI_ERR.

## InjectSigsegv

```
int InjectSigsegv( JNIEnv *env )
```

**Description**
Changes the facade pointer for

```
hpi_system_interface->GetMilliticks(sysGetMilliTicks)
```

to point to a routine that generates a SIGSEGV the next time it is called. This is intended only for testing purposes, to inject an error condition into the JVM, which results in JVM termination

**Parameters**

- Pointer to a JNIEnv.

**Returns**
JNI Return code JNI_OK if facade is successfully updated; otherwise, JNI_ERR.

## InjectOutOfMemory

```
int InjectOutOfMemory( JNIEnv *env )
```

**Description**
Changes the facade pointers for hpi_memory_interface->Malloc and hpi_memory_interface->Calloc to point at routines that will return a NULL pointer. This is intended only for testing purposes, to inject an out of memory error condition into the JVM.

**Parameters**

- Pointer to a JNIEnv.

**Returns**
JNI Return code JNI_OK if facade is successfully updated; otherwise, JNI_ERR.

## GetComponentDataArea

```
int GetComponentDataArea( JNIEnv *env, char *componentName,
       void **dataArea, int *dataSize )
```

**Description**
Gets the address and size of the global data area for the required JVM component. The component is identified by two characters. Valid components are 'ci', 'cl', 'dc', 'dg', 'lk', 'xe', 'xm' and 'st'.

**Parameters**

- Pointer to a JNIEnv.
- Component name.
- Pointer to the component data area.
- Size of the data area.

**Returns**
JNI Return code JNI_OK if valid component information is available; otherwise, JNI_ERR.

## SetOutOfMemoryHook

```
int SetOutOfMemoryHook( JNIEnv *env, void (*rasOutOfMemoryHook)
        (void) )
```

**Description**

Registers a callback function for an out-of-memory condition. It is called when `xmPanic` is called with a `PANIC_OUT_OF_MEMORY` reason code.

**Parameters**

- Pointer to a JNIEnv.
- Pointer to callback function.

**Returns**

JNI Return code `JNI_OK` if facade is successfully updated; otherwise, `JNI_ERR`.

## 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**

- 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.

## DynamicVerbosegc

```
void JNICALL *DynamicVerbosegc (JNIEnv *env, int vgc_switch,
             int vgccon, char* file_path, int number_of_files,
             int number_of_cycles);
```

**Description**

Switches verbosegc on or off dynamically.

**Parameters**

- 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 = -verbosegc, 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 verbosegc per file

**Returns**

None.

## 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 321). This function was added in the JVMRAS_1_3 interface.

**Parameters**

- Pointer to a JNIEnv.

**Returns**

None.

## 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 321). This function was added in the JVMRAS_1_3 interface.

**Parameters**

- Pointer to a JNIEnv.

**Returns**

None.

## GenerateHeapdump

```
int GenerateHeapdump( JNIEnv *env )
```

**Description**

Generates a Heapdump file.This function was added in the JVMRAS_1_3 interface.

**Parameters**

- Pointer to a JNIEnv.

**Returns**

JNI Return code `JNI_OK` if running dump is successful; otherwise, `JNI_ERR`.

# 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 system property **ibm.dg.trc.external** to route trace information to an external trace listener.

### The ibm.dg.trc.external property

The format of this property is:

`ibm.dg.trc.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 TraceFormat.dat to format JVM-generated tracepoints that are captured by the agent. TraceFormat.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:

**Type 0** Event

**Type 1** Exception

**Type 2** Entry

**Type 4** Exit

**Type 5** Exit-with-Exception

**Type 6** Mem

Any other type is reserved for development use; you should not find any on a retail build. Note that this condition is subject to change without notice. The version number will be different for each change.

# Chapter 36. Using the JVMPI

The JVMPI is a 2-way interface that allows communication between the JVM and a profiler. JVMPI allows third parties to develop profiling tools based on this interface. The interface contains mechanisms for the profiling agent to notify the JVM about the kinds of information it wants to receive as well as a means of receiving the relevant notifications. Several tools are based on this interface, such as Jprobe, OptimizeIt, TrueTime, and Quantify. These are all third-party commercial tools, so IBM cannot make any guarantees or recommendations with regard to their use. IBM does provide a simple profiling agent, based on this interface called **HPROF**.

## The HPROF profiler

**HPROF** is a profiler shipped with the IBM SDK that uses the JVMPI 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 should know what parts of the code are using large amounts of memory and CPU resources. HPROF is one of the nonstandard options 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.

**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.

**monitor=y | n**
The **monitor** option can help you understand how synchronization affects the performance of your application. Monitors are used to 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).

**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.

**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. This can be an issue 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".

## 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 Local.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:

- 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 actual 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 actual 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 11504e8). Looking further through the output you can see this class definition and name. It turns out to be the Object class (a class that every class inherits from). The JVM loads the entire superclass hierarchy before it can use a subclass. Thus, class definitions for all superclasses are always present. There are also entries for Objects (OBJ) and Arrays (ARR):

```
OBJ 111a9e78 (sz=60, trace=1, class=java/lang/Thread@8b0c38)
 name  111afbf8
 group  111af978
 contextClassLoader 1128fa50
 inheritedAccessControlContext 111aa2f0
 threadLocals 111bea08
 inheritableThreadLocals 111bea08
ARR 8bb978 (sz=4, trace=2, nelems=0, elem type=java/io/ObjectStreamField@8bac80)
```

If you set the **heap** option to ″sites″ or ″all″ (″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 on the CPU. If **cpu** is set to samples, you get an output containing 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 37. Using DTFJ

The Diagnostic Tooling Framework for Java (DTFJ) is a Java application programming interface (API) from IBM used to support the building of Java diagnostics tools.

DTFJ works with system dumps produced on a number of platforms; see "Which JVMs are DTFJ enabled?."

The system dumps generated by a JVM must be processed by the jextract tool and then you pass the resultant sdff file to DTFJ. For z/OS, you pass the generated dump file directly to DTFJ.

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:

- "Which JVMs are DTFJ enabled?"
- "Overview of the DTFJ interface" on page 376
- "DTFJ example application" on page 379

The full details of the DTFJ Interface are provided with the SDK as Javadoc in sdk/docs/dtfj.zip. DTFJ classes are accessible without modification to the class path.

## Which JVMs are DTFJ enabled?

For Java 1.4.2 Service Refresh 4 from IBM and above, the following platforms are supported:

- AIX PPC 32-bit
- Linux IA 32-bit
- Linux PPC 32-bit
- zLinux 32-bit
- z/OS 32-bit
- Windows IA 32-bit

# 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 378 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;

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.sov.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 at least one file. The files must be the dump itself and (optional and with the guidance of IBM service personnel) 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.IOException: Currently can only handle SDFF files, which _MUST have a suffix of ".sdff"_
or svcdumps
        at com.ibm.dtfj.image.sov.ImageFactory.getImage(ImageFactory.java:79)
        at com.ibm.dtfj.image.sov.ImageFactory.getImage(ImageFactory.java:171)
        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 378.

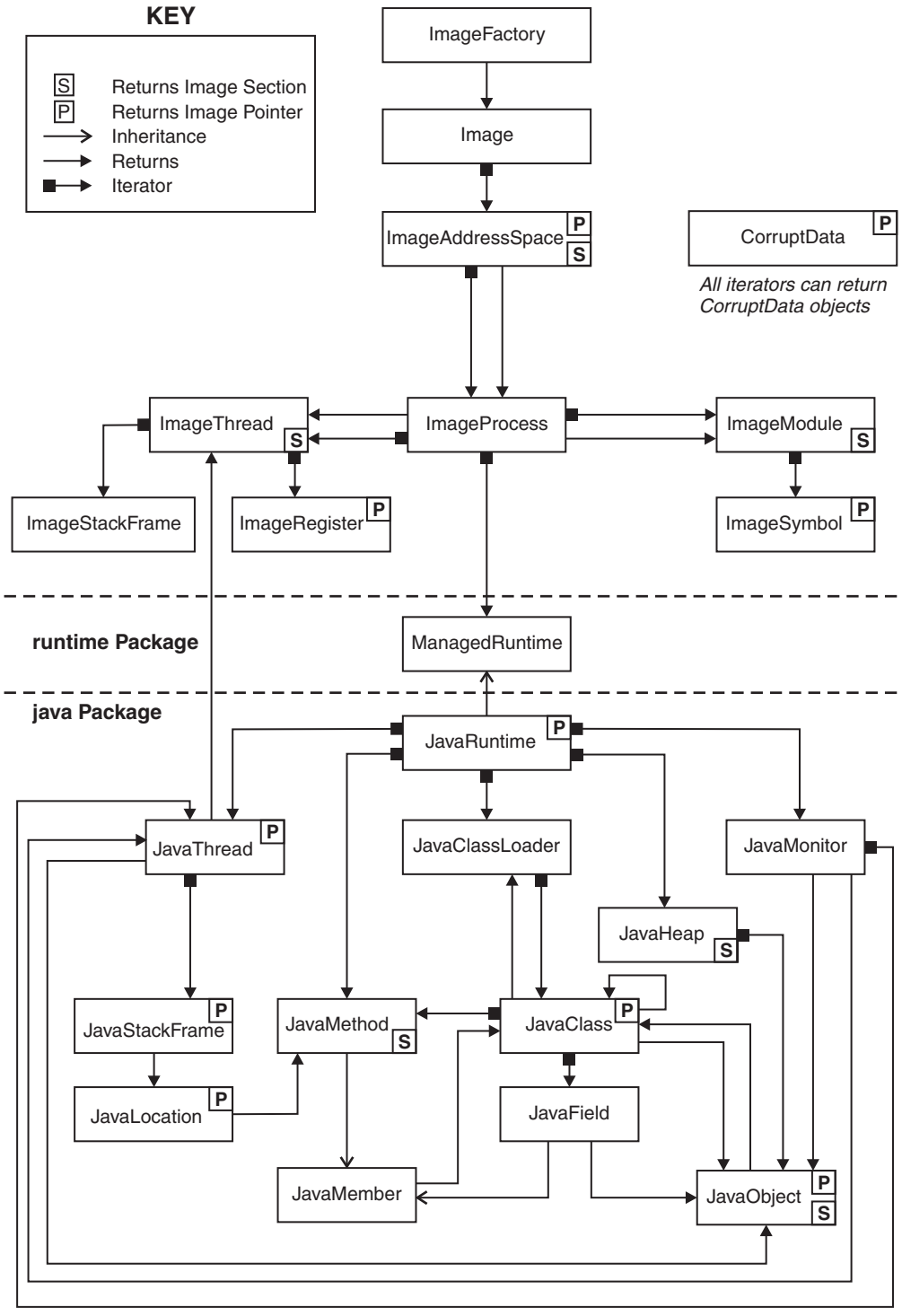



*Figure 13. Diagram of the DTFJ interface*

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 376.

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.sov.ImageFactory");
                ImageFactory factory = (ImageFactory) factoryClass
                        .newInstance();
                image = factory.getImage(f);
            } catch (Exception ex) { /*
                                      * Should use the error handling as
                                      * shown in DTFJEX1.
                                      */
                System.err.println("Error in DTFJEX2");
                ex.printStackTrace(System.err);
            }
```

```
        } else {
            System.err.println("No filename specified");
        }

        if (null == image) {
            return;
        }

        MatchingThreads(image);
    }

    public static void MatchingThreads(Image image) {
        ImageThread imgThread = null;

        Iterator asIt = image.getAddressSpaces();
        while (asIt.hasNext()) {
            System.out.println("Found ImageAddressSpace...");

            ImageAddressSpace as = (ImageAddressSpace) asIt.next();

            Iterator prIt = as.getProcesses();

            while (prIt.hasNext()) {
                System.out.println("Found ImageProcess...");

                ImageProcess process = (ImageProcess) prIt.next();

                Iterator runTimesIt = process.getRuntimes();
                while (runTimesIt.hasNext()) {
                     System.out.println("Found Runtime...");
                     JavaRuntime javaRT = (JavaRuntime) runTimesIt.next();

                    Iterator javaThreadIt = javaRT.getThreads();


                    while (javaThreadIt.hasNext()) {
                        Object tempObj = javaThreadIt.next();
                        /* Should use CorruptData
                         * handling for all iterators
                         */
                        if (tempObj instanceof CorruptData) {
                            System.out.println("We have some corrupt data");
                        } else {
                            JavaThread javaThread = (JavaThread) tempObj;
                            System.out.println("Found JavaThread...");
                            try {
                                imgThread = (ImageThread) javaThread
                                        .getImageThread();

                                // Now we have a Java thread we can iterator
                                // through the image threads
                                Iterator imgThreadIt = process.getThreads();

                                while (imgThreadIt.hasNext()) {
                                    ImageThread imgThread2 = (ImageThread) imgThreadIt
                                            .next();
                                    if (imgThread.equals(imgThread2)) {
                                        System.out.println("Found a match:");
                                        System.out.println("\tjavaThread "
                                                + javaThread.getName()
                                                + " is the same as "
                                                + imgThread2.getID());
                                    }
                                }
                            } catch (CorruptDataException e) {
                                System.err.println("ImageThread was corrupt: " + e.getMessage());
                            }
```

```
                        }
                    }
                }
            }
        }
    }
}
```

Many DTFJ applications will follow much the same model.

ll pop over.

# Chapter 38. Using third-party tools

This chapter introduces third-party diagnostic tools, in:



The tools that are described in this chapter have proved useful to the IBM Java service team. The team does not support the tools. The tools are presented in alphabetical sequence, and IBM does not recommend any of these tools in preference to other tools or give any warranty for them.

This chapter provides brief descriptions of the tools, the conditions in which you might find the tools useful, and brief instructions on how to use them. For more detailed information, refer to the documentation that is provided with the tools.

## GlowCode

GlowCode is a complete diagnostic and performance tool for C++ and applications that are running on Windows involving JNI. You can attach a running application to GlowCode to detect memory and resource leaks, isolate performance bottlenecks, profile and trace program execution, and find unexecuted code.

### Supported platforms

- Windows NT 4.0
- Windows 2000
- Windows XP
- Windows 95 and its derivatives
- Windows 98 and ME, are not supported

### Applicability

- Leaks
- Performance

### Summary

GlowCode provides a complete set of diagnostic tools.

The Profile tool provides a real-time hierarchical view of your program's execution. This tool gives you a time-aware overview of your program's execution.

The Report tool provides a real-time file, function, and line summary of your program's execution. This tool helps you find function bottlenecks quickly.

The Trace tool provides a real-time sequential view of your program's execution. This tool can insert trace statements into your running program's code to allow you to watch important variables. This tool also shows thread changes that can help you find thread synchronization bottlenecks.

The Profile, Report, and Trace tools use hooks that you can install in your application as it runs. You can selectively install hooks at a granularity of module, source file, function, or source code line, by using debug information in a .PDB file or exported symbol information in a .DLL file. Go to http://www.glowcode.com/summary.htm to view Hooks screenshots.

An Execution Analysis Details window is provided in the Profile, Report, and Trace tools. The Execution Analysis Details window lists all callers and callees of a function, along with count and time metrics. Go to http://www.glowcode.com/summary.htm to view screenshots of the Details tool.

The Memory tool:

- Uses counter hooks (that are automatically installed on the runtime heap) and important Win32 resources when GlowCode attaches itself to your application.
- Provides a real-time summary of allocations, and an expandable tree view of allocation details including the call stack that is active during each allocation. Go to http://www.glowcode.com/summary.htm to view Memory screenshots.
- Includes a heap leak detector, which, at your request, will recursively search heap and static memory for unreferenced heap allocations. Go to http://www.glowcode.com/summary.htm to view Leaks screenshots.

With GlowCode, you can zero in on an application method that might be causing a possible memory leak. Select the memory tab in the GlowCode window and click the **Leaks** button. This displays a Leak Result dialog that lists functions that might be responsible for memory leaks. These methods are a possible cause of memory leaks or memory consumption.

For more details about GlowCode, go to http://www.glowcode.com

### Running GlowCode

1. Open the GlowCode window that lists all the processes running on a system.
2. Select a process.
3. Click the **Attach** button.

## Heap analysis tool (HAT)

This tool relies on a JVMPI plug-in to profile the heap. The plug-in (start it with the JVM command-line option **-Xhprof**) creates a dump that you can analyze with Heap Analysis Tool (HAT). The plug-in is provided as part of the IBM SDK.

Download HAT from http://java.sun.com/people/billf/heap/index.html. Note that JVMPI (the profiling agent) is an IBM-supplied product that is maintained by the Java team.

HAT enables you to read and analyze the -Xhprof files. HAT helps to debug applications that unnecessarily retain objects (that is, they have memory leaks) by providing a convenient means to browse the object topology in a heap snapshot. HAT also allows you to trace the references to a given object from the rootset.

## Applicability

- Leaks
- Performance

## Generating a .hprof file

To generate a .hprof file:

- Run Java with the **-Xhprof** flag: `java -Xhprof:file=dump.hprof,format=b <class name>`
- To add a Heapdump to a .hprof file:
  - On *nix platforms, press Ctrl+V.
  - On Windows, press Ctrl+BREAK.

  When the JVM receives these signals, it writes a Heapdump file in its working directory.

## Running the program

HAT runs a Web server on your computer. Connect to it with your favorite browser.

To run the HAT server program, use the run shell script that is provided in the bin directory within hat.zip. Download hat.zip from http://java.sun.com/people/billf/heap/index.html.

```
./hat -port=7002 <.hprof file>
```

If you do not specify a port, the default is 7000. Point your browser to http://localhost:7000/.

The HAT server program compares two Heapdumps. You can specify a numbered Heapdump in the file. For example, to compare the first and second Heapdump in slotCar.hprof, use this command: ./hat -baseline= `<application name>.hprof:1 <application name>..hprof:2`

The heap profile file that you give to HAT must be in binary format. Use the **format=b** modifier on the -Xhprof command line. For example, to see all hprof options:

```
java -Xrunhprof:help
```

Use **hprof** to get information about the heap. You can profile the following with respect to heap:

- Dump -> Gives a dump of the heap. This will have the following distinctions:
  - ROOT : Root set by the Garbage Collector
  - ARR : Arrays
  - CLS : Classes
  - OBJ : Instances
- Sites -> Gives a dump of the allocation sites.
- All -> Gives the heap dump and the sites.

When the profiler has the output, run HAT on the profiler: `hat myClass.hprof`.

The output of this analysis will be available as an html page on localhost. To view the report, go to http://localhost:7000/. With this report, you can see the members of the rootset, that can give information about static references, JNI local references,

JNI global references, and system class references. You can view the instance counts for all the classes through a difference form in a generated report. The information about memory used and number of bytes for classes can help in finding the memory leak and improving the performance of your application.

## HeapWizard

The HeapWizard utility program is a product of IBM WebSphere L2 Support. This program is provided "as is", without guarantee or warranty of suitability of any sort. If you have questions about this product, or want to report a problem or suggest a new feature, e-mail ronv@us.ibm.com. For a copy of the tool, e-mail jvmcookbook@uk.ibm.com

**Note:** Currently, this utility works only on IBM JDK heapdump<pid>.<ts>.txt files. Future versions might handle other manufacturer's heap dumps.

HeapWizard reads and analyzes a heapdump.txt file. It can run from the command-line (to provide an XML format file) or work as a GUI to provide an expandable tree view. The suggested method of operation is the GUI. To output the results in XML format you need Xerces v1.4 or later.

To start HeapWizard in GUI mode, use the following:

```
java -Xms128M -Xmx512M -jar HeapWizard.jar
```

In most cases, you want to examine heapdumps one at a time. You can do this if you have two heapdumps from the same JVM instance; you can subtract an earlier (base) heapdump from a later one to get a heap difference.

### Terms

**Tree** A parent-child reference structure where a parent object holds references to one or more child objects that, in turn, might hold references to other objects, and so on.

**Note:** Where two or more objects reference each other (for example, X->A->B->C->A), HeapWizard breaks the loop at the last reference. Therefore, an object might appear to be at the top of a subtree although another object is referencing it in the heap.

**Root-level**
The top level of a tree. An object at the root-level does not have any other objects holding a reference to it.

**Rootsize**
The accumulated size of a root-level object's tree. This value is counted for root-level objects only.

### Heap view

HeapWizard displays two different perspectives of the Heapdump, Objects, and Classes. The Objects view, walks the tree of each root-level object and displays the objects to which the tree makes reference, subsequent referenced objects, and the accumulated size going down the tree.

The Classes view displays a flat list of all object types (classes) that were found in the Heapdump. It lists the total number of objects of that type, the accumulated subtree sizes, and the total accumulated size of root-level objects of that class type.

You should:

1. Check the Classes count= value. If you have more than 4000, it is probably suspect. For example, cases have occurred in which 20,000 JSP classes were displayed because a customer was dynamically generating JSPs with a script at the rate of a couple hundred per hour.
2. Check the first few rootsize values under Classes. This is probably the most useful value. Usually, a memory leak is caused when a single type of object is allocated repeatedly with a global reference that does not get cleared.
3. If step 2 shows that a generic class (such as java/util/Hashtable) is holding a large amount of memory, look in the Objects view to expand a few objects of the offending class type to see what objects it is holding. This helps you identify from where the objects are coming.
4. Generally, the finalizer thread is not very likely to be the cause of the memory leak unless it shows up repeatedly (across different heapdumps) with the same objects below it. The finalizer thread is where complex objects are cleaned up. So, on a very busy system, you might see many objects in the finalizer thread.

## Command-line options

Most of the options that are show here allow you to run HeapWizard without the GUI.

Usage:

```
java [jvm-options] HeapWizard [<infile>] [-<options>]
```

where <infile> is input heapdumpXXXXX.txt file and -<options> is one or more of the following:

**-noview**
Disable opening the GUI to display output.

**-xmlout=<xmlfile>**
Write xml output to file <xmlfile>.

**-base=<heapdump>**
Subtract <heapdump> as the baseline for <infile>.

**-encoding=<enc>**
Input file encoding name (default to system default).

**-trace=<level>**
Enable trace statements to stderr:

- 0 = Silent mode
- 1 = Show each phase
- 2 = Show object counts and memory usage (default)
- 3 = Level 2 plus show object names as tracing

**-classBySize=<len>**
The maximum number of classes that are displayed in the Classes By Size GUI tree (default 100).

**-objectBySize=<len>**
The maximum number of object trees that are displayed in the Objects By Size GUI tree (default 500).

**-noClassList**
Disable output of class name list.

**-jdk118**
Heapdump is from a 1.1.8 SDK.

**-help**
Display this listing.

**Note:** If -noview is set, and -xmlout is not set, XML output is to stdout.

**Examples:**

```
java -Xms32M -Xmx512M -jar HeapWizard.jar
java -Xms32M -Xmx512M -jar HeapWizard.jar heapdump2.txt -trace=3
-base=heapdump1.txt
java -Xms32M -Xmx512M -jar HeapWizard.jar heapdump1.txt -xmlout=heapdump1.xml -noview
```

# Jinsight

Jinsight is an IBM tool that is freely available. It is not a product of the IBM Java Technology Center or part of the SDK and is supported by its authors only. See the Jinsight Web site, referred to later in this section, for communication with this tool's support team.

## Supported platforms

Windows, AIX, and OS/390

## Applicability

- Leaks
- Performance

## Summary

Jinsight is a tool for analyzing and visualizing how Java programs execute. Jinsight has two parts:

- Java instrumentation, which generates trace data as your Java program runs.
- Jinsight visualizer, which reads the trace data and presents views for analysis.

Jinsight can be used:

- In scenarios where a big application is consuming a large amount of memory which results in "out of memory" exceptions or abnormal usage of memory.
- To detect memory leaks.
- To find the root cause of memory consumption.
- To analyze performance problems like CPU utilization.

## Jinsight views

The views of Jinsight outputs much application-related information like:

- Methods that are frequently called while the application and also the constructors are executing.
- Objects created in the sequence of application execution (in the form of trees) and also the total memory usage for those objects. These objects include class/static objects, JNI references, Java Objects, and Java stacks.

Here are the views that Jinsight outputs, and their usage.

**Execution view**
Explore the detailed program execution sequence per thread.

**Execution pattern view**
Browse recurring execution patterns in aggregated form.

**Table view**
See summary information about a run in tabular form. Table views exist for classes, objects, methods, invocations, threads, packages, and user-defined slices.

**Call tree view**
Study summary statistics for call paths that lead from or to selected method invocations.

**Object histogram view**
Examine instances of objects that are grouped by class, and their level of activity.

**Method histogram view**
Examine methods that are grouped by class, and their level of activity.

**Reference pattern view**
Explore patterns of references to or from a set of objects, in varying detail; this is useful for studying data structures and finding memory leaks.

You can use various Jinsight functions to answer questions such as:

- What are calling relationships among a group of methods?
- Why is a method taking so long?
- Why is a method being called?
- Where are all the places in which a method is called?
- What is the context of each individual method invocation?
- Are repetitive sequences in summary form or individual?
- Which are expensive methods?
- Which methods are taking a long time?

Also for analyzing memory related problem like memory leaks or huge memory consumption, look for the following details:

- Look for activity in constructors.
- Look for the objects that are created the most frequently.
- Find the objects that create the objects that you are studying.

For more details, refer to http://www.alphaworks.ibm.com/tech/jinsight.

## Running Jinsight

To use Jinsight, do these two steps:

1. Generate the Application Trace data. Set `CLASSPATH=<path where your program is>`. Run Jinsight with your application as a parameter:

   ```
   jinsight_trace -tracemethods <yourProgram> <yourProgramArgs>
   ```

   A trace file yourprogram.trc is created as your program runs.
2. Take the snapshot of the Memory Heap with `set CLASSPATH=<path where your program is>`. Run JInsight: **jinsight_trace <yourProgram> <yourProgramArgs>**

To take a snapshot of all the objects, or the objects and their references during an execution:

- On *nix platforms, press Ctrl+\.

- On Windows, press Ctrl+BREAK.

You will see a prompt:

```
trace:yourprogram.trc, Tracing status: CORE
---------------------------------------------
Jinsight Tracing Options:
[tm]          - start tracing methods
[sm]          - stop tracing methods
[tp]          - start tracing population events
[sp]          - stop tracing population events
[S]           - stop tracing
[g]           - force synchronous garbage collection
[dp]          - dump a generation's population
[dr]          - dump a generation's references
[ps]          - print system state
[q]           - quit program
[u <string>]  - user event
[<Enter>]     - exit command handler
---------------------------------------------
>
```

Use the commands above on the command line as required.

## Visualizing an application trace

1. Run jinsight.bat, and specify a trace file as a parameter: **jinsight GCExample1.trc**.
2. The Jinsight workspace window is displayed.
3. Click on the **Load** button for loading the trace file.

---

# JProbe

The suite of JProbe products consists of:

- The JProbe Profiler with memory debugger
- JProbe ThreadAnalyser
- Jprobe Coverage

## Applicability

- Leaks
- Hangs

## Supported platforms

Windows, Linux, and AIX

## Summary

With **JProbe ThreadAnalyser**, you can identify possible deadlocks, data races, and stalled threads.

**JProbe Coverage** helps you to understand exactly what code your current tests cover, including how many times a line of code executes and what code is missed completely. You can then streamline your tests by focusing on the critical path, reducing redundancies, and improving coverage.

Using **Memory Debugger** you can trace objects that are consuming more memory (which can lead to out-of-memory conditions). Memory Debugger also traces the

objects that the Garbage Collector might not be able to reach and that could therefore cause memory leaks. The output is useful for analyzing the Java heap.

JProbe's Memory Debugger outputs memory-related information, such as:

- All the Java objects that are allocated with the name of the class, package, number of instances, and total memory occupied
- Referrers and references of all the objects
- All objects that have been garbage collected, live objects, and from where these objects are allocated
- A graphical representation of the heap that shows how the Java heap is growing, when the Garbage Collector is started, and the behavior of the heap
- When the Garbage Collector comes into action, the decrease in the count of objects
- Live objects being created whenever an application creates objects; for example, adding a dynamic button to a panel
- A snapshot of the heap at various instances of the application's execution
- Force garbage collection

By using the above functions, you can detect objects that might be consuming memory, or objects that the Garbage Collector might not be able to reach. Also, using JProbe's Thread Analyzer, you can detect possible deadlock conditions or data races that might lead an application to hang.

For more details on JProbe, see http://www.sitraka.com/software/jprobe.

### Using the Memory Debugger

1. Go to the JProbe LaunchPad.
2. In the **Program** Tab, give the details of the application.
3. In the **VM** Tab, give the details of Java VirtualMachine that you are using.
4. In the **Heap** and **Performance** tabs, select the information that you want to view.
5. Select **Run**.

## JSwat

JSwat is a graphical Java debugger. You can use it instead of **jdb** to debug any Java applications. JSwat is more useful for debugging your application than it is for debugging faults in the JVM. JSwat, should therefore help you to find bugs.

Typically, Java application programmers can use JSwat to debug their Java applications during the development stage. In large Java applications, you can use JSwat to narrow down to the failing area of code.

Documentation on JSwat is included with the distributable package of JSwat. Download it from http://www.bluemarsh.com/java/jswat/.

### Applicability

Use JSwat for Java source-level debugging for any kind of problem. It is particularly useful for identifying the area of Java code that is causing the problem.

## Summary

JSwat supports console and graphical modes of debugging. Console mode is more useful when the AWT/JFC classes are not provided in your runtime environment, or if you prefer a quicker interface for debugging.

JSwat debugger supports:

- Threads display
- Tree display of classes, including inner classes
- Local variables display
- Watchpoints display
- Stack frames display
- Source code viewer
- Breakpoints controlling
- Remote debugging
- Applet debugging
- Servlet debugging

## Preparing for JSwat debugging

To debug any Java application in JSwat, first compile your Java application with **-g** option (**javac -g Testcase.java**). In the case of a huge Java application, if the area of code to be debugged is known, only that part of the code can be compiled with **-g** option.

## Running your application in JSwat debugger

- Add the files<<sdk_home>\lib\tool.jar and <jswat_home>\jswat-20011027.jar to your **CLASSPATH**.
- Run your application in the JSwat debugger using the command `java com.bluemarsh.jswat.Main TestCase`.

If the source code of the application that is being debugged is not in the current directory, JSwat can locate the source code only if you use the option `-Djava.source.path=<directory of application>`. For example,

```
java -Djava.source.path=C:\nprashan\src com.bluemarsh.jswat.Main TestCase
```

# Process Explorer

This a Windows tool that runs on all Windows platforms. It is freeware and is available for download from http://www.sysinternals.com.

Process Explorer shows you information about the handles and DLLs processes that have opened or loaded.

The Process Explorer display consists of two sub-windows. The top window always shows a list of the currently active processes, including the names of their owning accounts. The information that is displayed in the bottom window depends on the mode that Process Explorer is in:

- If PE is in Handle mode, the handles that were opened by the process that was selected in the top window are displayed.
- If PE is in DLL mode, the DLLs and memory-mapped files that the process has loaded are displayed.

PE also has a powerful search capability that quickly shows which processes have particular handles opened or DLLs loaded.

The unique capabilities of Process Explorer make it useful for tracking down DLL-version problems or handle leaks, and provide insight into the way Windows and applications work. Here is a screenshot of Process Explorer Version 5.2.

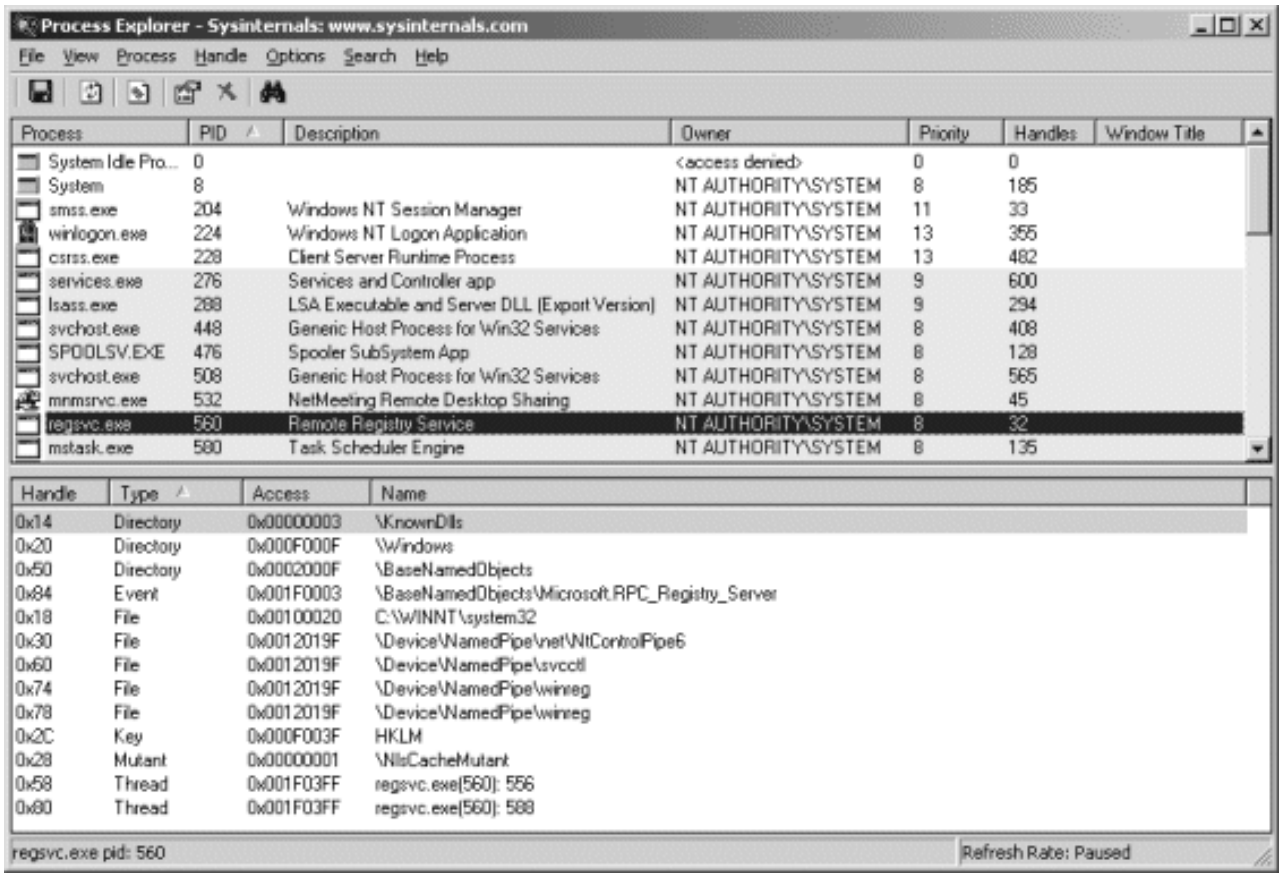



*Figure 14. Screenshot of Process Explorer*

To use PE:

1. Bring it up and select the process you want to view: **java.exe**.
2. From the **View** menu, select either **DLLs** or **handles**. The bottom window will refresh appropriately.
3. In **DLL** mode the **path** section of the display will show the JVM DLLs (they are located in the \bin directory of your Java path).

**Note:** If you are running the WebSphere Application Server, more than one **java.exe** process might be running.

# Part 5. Appendixes

# Appendix A. Compatibility tables

## WebSphere Application Server and JVM/SDK levels

Before release 5 of the IBM WebSphere Application Server, the SDK and the ORB levels did not match. This table shows the version of the embedded JVM that ships with the corresponding version of the WebSphere Application Server.

| WebSphere Application Server | SDK | ORB |
|---|---|---|
| 3.5.4 | 1.2.2 | 1.2.2 |
| 3.5.5 | 1.2.2 | 1.2.2 |
| 4.0.1 | 1.3.0 | 1.3.0 |
| 4.0.2 | 1.3.0 | 1.3.0 |
| 4.0.3 | 1.3.1 | 1.3.0 |
| 4.0.4 | 1.3.1 | 1.3.0 |
| 4.0.5 | 1.3.1 | 1.3.1 |
| 5.0 | 1.3.1 | 1.3.1 |
| 5.0.1 | 1.3.1 | 1.3.1 |
| 5.0.2 | 1.3.1 | 1.3.1 |
| 5.1 | 1.4.1 | 1.4.1 |
| 5.1.0 | 1.4.1 Service Refresh 1 | 1.4.1 |
| 5.1.1 | 1.4.2 | 1.4.2 |

# Appendix B. ORB tracing for WebSphere Application Server version 5

The diagnostic trace configuration settings for a server process determine the initial trace state for that server process. The configuration settings are read at server startup and are used to configure the trace service, either at server startup or while the server is running. You can select whether to enable or disable ORB trace, and you can change many of the trace service properties or settings while the server process is running. This appendix describes:

- "Enabling trace at server startup"
- "Changing the trace on a running server" on page 400
- "Selecting ORB traces" on page 400

For more information, see:

ftp://ftp.software.ibm.com/software/webserver/appserv/library/wasv5base_pd.pdf
or
http://www-3.ibm.com/software/webservers/appserv/was/library/
or the WAS problem determination guide:
http://www.redbooks.ibm.com/abstracts/sg246798.html

## Enabling trace at server startup

The diagnostic trace configuration settings for a server process determine the initial trace state for that server process. The configuration settings are read at server startup and are used to configure the trace service. To enable the trace:

1. Start the AdminConsole.
2. In the console navigation tree, click **Troubleshooting > Logging and Tracing**.
3. Click **Server > Diagnostic Trace**.
4. Click **Configuration**.
5. Select the Enable Trace check box to enable trace, or clear the check box to disable trace.
6. Set the trace specification to the desired state by entering the correct TraceString:

   ```
   ORBRas=all=enabled
   ```

7. Select whether to send trace output to a file, or to an in-memory circular buffer.

   If you select a file, go to step 8.

   If you select an in-memory circular buffer, go to step 11.
8. If you have selected a file for trace output, set the maximum size in MB to which the file is allowed to grow. When the file reaches this size, the existing file is closed, renamed, and a new file with the original name is opened. The new name of the original file is the original name with a timestamp qualifier added to it.
9. Specify how many history files you want to keep.
10. Go to step 12 on page 400.
11. If you have selected an in-memory circular buffer for the trace output, set the size of the buffer. The size of the buffer determines the maximum number of entries that are to be kept in the buffer at any given time. Specify the size of

the buffer in thousands of entries. For example, if you want 1000 entries, specify **1**; if you want 3000 entries, specify **3**.

12. Select the desired format for the generated trace.
13. Save the changed configuration.
14. Start the server.

## Changing the trace on a running server

You can change the trace service state that determines which components are being actively traced for a running server. To do this:

1. Start the AdminConsole.
2. In the console navigation tree, click **Troubleshooting > Logging and Tracing**.
3. Click **Server > Diagnostic Trace**.
4. Select the Runtime tab.
5. If you want to write your changes back to the server configuration, select the Save Trace check box.
6. Change the existing trace state by changing the trace specification to the desired state.
7. If you want to change from the existing trace output, configure a new one.
8. Click **Apply**.

## Selecting ORB traces

You can select to enable or disable the ORB traces. To do this:

1. Start the AdminConsole.
2. In the console navigation tree, click **Servers > Application Server**.
3. Click **Server**.
4. Click **Configuration**
5. In the Additional Properties panel, click **ORB Service**.
6. Select the Enable Trace check box to enable ORB trace, or clear the check box to disable ORB trace.
7. If you have chosen to disable ORB trace, go no further with these instructions.

   If you have chosen to enable ORB trace, go to the next step.
8. In the Additional Properties panel, select **Custom Properties**.
9. Ensure that these two property names and values are present:

   ```
   com.ibm.CORBA.Debug , true
   com.ibm.CORBA.CommTrace , true
   ```
10. Add them if they are not present.

# Appendix C. CORBA GIOP message format

Table 29 shows:

- All types of messages
- The values of those messages as an integer number
- Whether those messages contain only a header or a header and a body
- Whether those messages are supported in GIOP versions 1.0, 1.1, and 1.2

*Table 29. CORBA GIOP messages*

| Message | Value | Header | Body | 1.0 or 1.1 supported | 1.2 supported |
|---|---|---|---|---|---|
| Request | 0 | X | X | X | X |
| Reply | 1 | X | X | X | X |
| Cancel Request | 2 | X | | X | X |
| Locate request | 3 | X | | X | X |
| Locate reply | 4 | X | X | X | X |
| Close connection | 5 | | | X | X |
| Message error | 6 | | | X | X |
| Fragment | 7 | X | X | | X |

**Note:** From now and on in this chapter, each cell (table column) represents 1 byte unless specified otherwise. Alignment of fields is not specified in the following byte description. In a GIOP message, some fields need to start at a 4- or 8-byte boundary. Extra bytes of padding are present (always set to 0).

## GIOP header

All types of messages that are described in Appendix C, "CORBA GIOP message format" start with the GIOP Message Header:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 47=G | 49=I | 4F=0 | 50=P | Major; for example, 01 | Minor; for example, 02 | Flags | Value | (4 bytes) length of the rest of the message |

In GIOP1.0, the least-significant bit of the Flags byte (that is, the first bit on the right of the byte) indicates the byte sequence (big endian or little endian). In GIOP 1.1 and 1.2, the least-significant bit indicates the byte sequence that is used in later elements of the message. A value of false (0) indicates a big-endian byte sequencing; true (1) indicates little-endian byte sequencing.

The bit that is immediately to the left of the least-significant bit indicates whether or not more fragments follow. A value of false (0) indicates that this message is the last fragment; true (1) indicates that more fragments follow this message. The

most-significant six bits are reserved; for GIOP 1.2 (1.1) they must be set to zero. The Value field is the field that is indicated in Appendix C, "CORBA GIOP message format," on page 401.

## Request header

| | | | | |
|---|---|---|---|---|
| Request id (4 bytes) | Response Expected | Reserved | Reserved | Reserved |

The Response Expected flag indicates whether this request expects a reply from the server. Values 1 = WITH_SERVER and 3 = WITH_TARGET correspond to a true value. Therefore, the client expects a reply. A value of 0 = NONE or WITH_TRANSPORT means that no reply is required. The reserved bytes are for future use.

After this first 8 bytes, the header continues with the specification of the remote reference. This specification, however, differs in different version of the GIOP. In GIOP 1.0 and 1.1, the specification is:

| | |
|---|---|
| Length of the Object Key (4 bytes) | Object key (see previous length in bytes) |

In GIOP 1.2, the specification is more complex. The next value of Addressing Disposition index decides whether to insert an object key, a profile, or a full IOR (one row corresponds to one value):

| | | | |
|---|---|---|---|
| Addressing disposition (2 bytes):<br><br>0=object key<br>1=profile<br>2=IOR | Object key length (4 bytes) | Object key | |
| | IOR profile ID (4 bytes) | IOR profile length | IOR profile data |
| | IOR profile index | Full IOR | |

Then for all versions of GIOP the header continues with:

| | | |
|---|---|---|
| Length of operation name | Operation name | N= Number of service contexts present (4 bytes) |

and a sequence of N service contexts must come next. The following describes how one of these service contexts is written. N of them are written consecutively.

| | | |
|---|---|---|
| Service context ID (4 bytes) | Service context length (4 bytes) | Service context data |

## Request body

| | |
|---|---|
| Marshaled parameters (CORBA valuetype) | Context pseudo object (for GIOP 1.0/1.1 only) |

## Reply header

For GIOP 1.2:

| | |
|---|---|
| Request ID (4 bytes) | Reply status (4 bytes) |

The reply status can be:
- 0 = NO_EXCEPTION
- 1 = USER_EXCEPTION
- 2 = SYSTEM_EXCEPTION
- 3 = LOCATE_FORWARD
- 4 = (deprecated)
- 5 = NEEDS_ADDRESSING_MODE (GIOP 1.2 only)

Request ID and reply status are then followed by:

| | |
|---|---|
| Number of service contexts (4 bytes) | < sequence of service contexts as before > |

**Note:** In GIOP 1.0/1.1, the request ID and the reply status comes after the service context list.

## Reply body (based on reply status)

- NO_EXCEPTION:

| |
|---|
| Marshaled parameters |

- USER_EXCEPTION: Varies (see CORBA specification)
- SYSTEM_EXCEPTION:

| | | | |
|---|---|---|---|
| Exception ID length (4 bytes) | Exception ID | Minor code (4 bytes) | Completion status (4 bytes) |

- LOCATE_FORWARD:

| |
|---|
| IOR (starts with type ID) |

- NEEDS_ADDRESSING_MODE (GIOP 1.2 only):

| |
|---|
| Addressing Disposition (2 bytes) |

## Cancel request header

This contains only the request ID coded in 4 bytes.

## Locate request header

| | | | | |
|---|---|---|---|---|
| Request ID | Addressing disposition (GIOP 1.2 only) | Object key length (4 bytes) | Object key | |
| | | IOR Profile ID (4 bytes) | IOR profile length | IOR profile data |
| | | IOR profile index | Full IOR | |

GIOP 1.0/1.1 supports only the object key version (first row only) and no addressing disposition is specified.

## Locate reply header

| Request ID (4 bytes) | Reply Status (4 bytes) |
|---|---|

## Locate reply body

- UNKNOWN_OBJECT = 0: No locate reply body
- OBJECT_HERE = 1: No locate reply body
- OBJECT_FORWARD = 2: IOR starting with the type ID
- Skip 3 (now deprecated)
- LOC_SYSTEM_EXCEPTION = 4: (Same as SYSTEM_EXCEPTION in reply body)
- NEEDS_ADDRESSING_MODE = 5: Addressing disposition index in two bytes (short)

## Fragment message

The fragment message observes these rules:

- The fragment length plus the GIOP header length (12 bytes) is a multiple of 8 for all but last message.
- All fragments must include at least the GIOP header and the request ID (total length 16 bytes).
- In the GIOP header of the first fragment, the message type can be request, reply, locate request, and locate reply. The fragment flag is set to 1.
- In the fragments that follow the first one, the message type is Fragment, and the fragment flag is set to 1, except in the last fragment where the flag is set to 0.

## Fragment header (GIOP 1.2 only)

The fragment header is made of only four bytes that represent the request ID.

# Appendix D. 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 191 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.DDMMYYY.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 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 E. Environment variables

This appendix provides the following information about environment variables:



## Displaying the current environment

To show the current environment, run:

set (Windows)
env (Unix)
set (z/OS)

To show a particular environment variable, run:

echo %ENVNAME% (Windows)
echo $ENVNAME (Unix)

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 30 on page 408 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.

*Table 30. 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. | |
| IBMJAVA_ INPUTMETHOD_ SWITCHKEY | See *SDK User Guide*. | Unix only. |
| IBMJAVA_ INPUTMETHOD_ SWITCHKEY_ MODIFIERS | See *SDK User Guide*. | Unix only. |
| 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 -Djava.compiler=NONE -verbose:gc | Any options are overridden by equivalent options that are specified when Java is started.<br><br>Does not support 'showversion'.<br><br>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 323. |
| IBM_USE_FLOATING_ STACKS | Enable the JVM to use the floating stacks | Linux only. |
| JAVA_ASSISTIVE | To prevent the JVM from loading Java Accessibility support, set the JAVA_ASSISTIVE environment variable to OFF. | |
| JAVA_DEBUG_PROG | JAVA_DEBUG_PROG=<prog_exe><br><br>This option starts the JVM through <prog_exe> executable. Usually used to debug; for example: JAVA_DEBUG_PROG=gdb | Linux only. |
| JAVA_FONTS | Define the font directory. | |
| 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 vesion of Mozilla | Unix only. |

*Table 30. 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. | Unix 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. | Unix 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. |
| PLUG_IN HOME | Set the Java plug-in path | AIX only. |
| SYS_LIBRARY_PATH | Specify the library path. | Linux and Unix only. |

*Table 31. Basic JIT options*

| Variable Name | Variable Values | Notes |
|---|---|---|
| IBM_MIXED_ MODE_THRESHOLD | Threshold for method compilation. This is the number of times a method or loop is executed before it is considered for compiling. A value of 0 means that the compiler attempts to compile every method on its first invocation. | The default values are around 500 through 1000, depending on platform. |
| JAVA_COMPILER | The runtime Java compiler to use. Default value is jitc, the Just-In-Time compiler. A value of NONE causes the Java bytecode to be interpreted only, not compiled. | Any unknown value behaves like NONE, and prints a message to say that the bytecode will be interpreted only. You can override this by setting the -Djava.compiler property on the command line. |
| JITC_COMPILEOPT | Specify debugging options to the JIT compiler. These include: COMPILING: Output which methods are compiled or are skipped while being compiled. SKIP: Skip compiling all methods. SKIP{P/C}{M}: Skip the M method from the C class in the P package. SKIP{P1/C1}{M1}{P2/C2}{M2}: Skip compiling both P1.C1.M1() and P2.C2.M2() methods. NALL: Syntax as with SKIP except that it disables only optimization of methods. The options can be combined with : (colon) on Unix and ; (semicolon) on Windows. | The C and M can be a wildcard specification. For example, **JITC_COMPILEOPT= COMPILING:SKIP{P1/*}{*} {P2/Q2/C2}{M2}:NALL{P3/C3}{*}** shows what is compiled, skips everything in package P1 and the P2.Q2.C2.M2() method, and disables optimization of every method in the P3.C3 class. |

*Table 32. Javadump and Heapdump options*

| Variable Name | Variable Values | Notes |
|---|---|---|
| DISABLE_JAVADUMP | Disables Javadump creation by setting to true (case-sensitive). | Can use command line parameter **-Xdisablejavadump** instead. Avoid using this environment variable because it makes it more difficult to diagnose failures.<br><br>On z/OS, **JAVA_DUMP_OPTS** should be used in preference. |
| IBM_HEAPDUMP or IBM_HEAP_DUMP | Enables heapdump. | See Chapter 26, "Using Heapdump," on page 245. |
| IBM_HEAPDUMP_ OUTOFMEMORY | Generates a heapdump when an out-of-memory exception is thrown. Set to any value. | Set to FALSE to turn off this option. |
| IBM_NOSIGHANDLER | Disables Java dump creation by setting to true. | AIX only. To be used in conjunction with the **DISABLE_JAVADUMP** option for builds subsequent to the January 2003 builds. |
| IBM_JAVACOREDIR | Specify an alternative location for Javadump files; for example, on Linux IBM_JAVACOREDIR=/dumps | On z/OS **_CEE_DMPTARG** is used instead. |
| IBM_JAVADUMP _OUTOFMEMORY | Trigger a Javadump on heap exhaustion/OutOfMemory error. Set to any value to turn this event trigger on. | |
| JAVA_DUMP_OPTS | Controls how diagnostic data are dumped. | Recommended default value is described in Chapter 12, "First steps in problem determination," on page 97. |
| _JVM_THREAD _DUMP_BUFFER_SIZE | Specify maximum size of Javadump file in bytes. | Default is 512 KB. |
| TMPDIR | Specify an alternative temporary directory. This is used only in the case when Javadumps and Heapdumps can be written only to a temporary directory. | Defaults to /tmp on Unix and \tmp on Windows. |

*Table 33. Diagnostics options*

| Variable Name | Variable Values | Notes |
|---|---|---|
| IBM_JVM_DEBUG_PROG | Launches the JVM under the specified debugger. | Linux only. |
| IBM_JVM_MONITOR_OLD | Allows you to restore the old algorithm when you are using SLES 8. See "Scheduler limitation on SLES 8" on page 145. | |
| IBM_MALLOCTRACE | Setting this variable to a nonnull value enables the tracing of memory allocation in the JVM. It is used with the MALLOC_TRACE environment variable. See http://www.gnu.org/manual/glibc-2.2.3 /html_node/libc_37.html for details of how to use MALLOC_TRACE. See "Tracing" on page 133. | Linux only. |

*Table 33. Diagnostics options (continued)*

| Variable Name | Variable Values | Notes |
|---|---|---|
| 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, `java/lang/InternalError` | |
| _JAVA_LAUNCHER _DEBUG | Specifying _JAVA_LAUNCHER_DEBUG=TRUE prints debug information on JVM environments and on loaded libraries while initializing the JVM. | Default is FALSE. Output is written to standard out. |
| JAVA_PLUGIN_TRACE | To take a Java plug-in trace, set JAVA_PLUGIN_TRACE=1 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. |
| JAVA_VM_PREWAIT | This option causes the JVM when acting as a plug-in to wait for 30 seconds at startup. This time could for example be used to attach a debugger before initialization starts. | Unix only. |
| LD_ASSUME_KERNEL | Can be set to 2.2.5 for Linuxthreads with fixed stacks. Can be set to 2.4.19 on RedHat or 2.4.21 on SuSE for Linuxthreads with floating stacks. | Controls which threading library is used. Use fixed stacks on RedHat distros with kernel levels 2.4.0 through 2.4.10. Otherwise, use the default (Linuxthreads on older distros, NPTL on newer distros) unless you are investigating a possible problem in the threading libraries. |
| ALLOCATION _THRESHOLD | Enables a user to identify the Java stack of a thread making an allocation request of larger than the value of this environment variable.<br><br>The output is:<br>`Allocation request for`<br>`<allocation request> bytes`<br>`<java stack>`<br><br>If there is no Java stack, `<java stack>` becomes `No Java Stack`. | |

## z/OS environment variables

**JAVA_DUMP_OPTS**

See Chapter 27, "JVM dump initiation," on page 251 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:

```
%s.JVM.TDUMP.&JOBNAME..D&YYMMDD..T&HHMMSS
```

where the hlq 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_PROPAGATE=NO**
Application programs and middleware products can use the JNI calls **AttachCurrentThread** and **DetachCurrentThread** to attach and detach respectively the current z/OS native thread to a JVM. These calls cause internal JVM data areas, that are associated with the thread, to be created and destroyed.

During normal thread creation in the JVM; that is, when a Java program creates a new thread, the security and workload (WLM) context that is associated with the parent thread is propagated to the new thread. In the case of a thread being attached to the JVM from native code, as described above, the thread is treated as a new parent. A token to its security and workload context is saved and propagated to any threads that this thread creates while running in the JVM.

This default behavior closes a possible security problem that is caused because, by default, when Java code creates a new thread, a new z/OS USS pthread is created without a security context associated with it. This means that the new pthread inherits the authority of the address space. This is not always desirable.

To support those cases in which middleware is handling security concerns, the environment variable **JAVA_PROPAGATE** can be set to *NO* before starting the JVM, therefore turning off the default propagation of context behavior. The environment variable **JAVA_PROPAGATE** is tested by the JVM code that is concerned with thread creation only during JVM initialization, to assure that this behavior cannot be modified by Java application code during JVM operation.

For performance reasons, the IBM WebSphere Application Server supports a nodetach mode of operation. In this case, threads are not detached from the JVM between separate units of work. In general, nodetach does not work with the JVM's default propagation because it is the DetachCurrentThread that causes the JVM to unset the osenv USS block on changing of thread identity. This means that if the middleware, for example the WebSphere Application Server, wants to change the userid that is associated with the thread between units of work (JVM invocations), it cannot.

When the WebSphere Application Server was initially available, this was the operation: the WebSphere Application Server did not use the default propagation behavior of the JVM, and so detach and nodetach both worked. In later releases the WebSphere Application Server began to use JVM propagation for the case where a client or surrogate id needed to be used for a thread accessing DB2, for example.

From WebSphere Application Server 4.0, the environment variable **JAVA_PROPAGATE** is set to *NO* before starting the JVM, so that propagation of context behavior is turned off.

**JAVA_THREAD_MODEL**
JAVA_THREAD_MODEL can be defined as one of:

**GREEN**
JVM uses the pthread replacement routines (referred to as green threads).

**NATIVE**
JVM uses the standard, POSIX-compliant thread model that is provided by the JVM. All threads are created as __MEDIUM_WEIGHT threads.

**HEAVY**
JVM uses the standard thread package, but all threads are created as __HEAVY_WEIGHT threads.

**MEDIUM**
Same as NATIVE.

**NULL**
Default case: Same as NATIVE/MEDIUM.

# Appendix F. Messages and codes

This chapter lists error messages in numeric sequence in:



These are messages, error codes, and exit codes that are generated by the JVM. You are unlikely to see these because the JVM generates them only when it finds an unrecoverable internal processing fault.

If the JVM fills all its 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; for example, "OUT OF MEMORY JVMCI003".

## Where do the messages appear?

Each message has a 'system action' description.

If the system action says that the JVM throws an exception, you see the message as part of the exception trace, which is directed to the stderr output stream. Therefore, you will see the message on the console or in a file that has captured stderr.

If the system action is not to throw an exception, the message appears on stderr unless otherwise stated.

## JVM error messages for JVMCI

**JVMCI001 OutOfMemoryError, allocating a JNI global ref**

**Explanation:** A call to jni_NewGlobalRef() has failed because not enough memory is available.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java -Xmx option.

**JVMCI002 OutOfMemoryError, stAllocObject for jni_AllocObject failed**

**Explanation:** A call to jni_AllocObject() has failed because not enough memory is available.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java -Xmx option.

**JVMCI003 OutOfMemoryError, stAllocArray for jni_NewString failed**

**Explanation:** A call to jni_NewString() has failed because not enough memory is available.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java -Xmx option.

---

**JVMCI004 OutOfMemoryError, stAllocObject for jni_NewString failed**

**Explanation:** A call to jni_NewString() has failed because not enough memory is available.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java -Xmx option.

---

**JVMCI005 OutOfMemoryError, dcUTF2JavaString failed**

**Explanation:** A call to jni_NewStringUTF() has failed because not enough memory is available.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java -Xmx option.

---

**JVMCI006 OutOfMemoryError, dcUnicode2UTF failed**

**Explanation:** A call to jni_GetStringUTFChars() has failed because not enough memory is available.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java -Xmx option.

---

**JVMCI007 OutOfMemoryError, stAllocArray for jni_NewObjectArray failed**

**Explanation:** A call to jni_NewObjectArray() has failed because not enough memory is available.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java -Xmx option.

---

**JVMCI008 OutOfMemoryError, eeGetFromJNIEnv failed**

**Explanation:** A call to jni_New##type##Array() has failed because not enough memory is available. ##type## can be any of: Boolean, Byte, Short, Char, Int, Long, Float, or Double.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java -Xmx option.

---

**JVMCI009 OutOfMemoryError, IBMJVM_NewArray - stAllocArray for new array failed**

**Explanation:** A call to IBMJVM_NewArray() has failed because not enough memory is available.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java **-Xmx** option.

---

**JVMCI010 OutOfMemoryError, sysMalloc failed**

**Explanation:** The invocation of a native method needed a buffer that is larger then the default preallocated 256 bytes, but was unable to get enough storage.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

---

**JVMCI011 OutOfMemoryError, can't create a new array**

**Explanation:** JVM_GetClassSigners failed to get storage by way of stAllocArray.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java -Xmx option.

---

**JVMCI012 OutOfMemoryError, stAllocArray failed**

**Explanation:** JVM_GetStackAccessControlContex failed to get storage by way of stAllocArray.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java -Xmx option.

---

**JVMCI013 OutOfMemoryError, create clone failed**

**Explanation:** Object.clone() failed to get storage for an object.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java -Xmx option.

**JVMCI014 OutOfMemoryError, stAllocArray failed**

**Explanation:** Object.clone() failed to get storage for an array.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java -Xmx option.

**JVMCI015 OutOfMemoryError, cannot create any more threads due to memory or resource constraints**

**Explanation:** JVM_StartThread() call to xmCreateThread() returned either SYS_NOMEM or SYS_NORESOURCE.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Either not enough resources are available to create a new threads, or the C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

**JVMCI016 OutOfMemoryError, stAllocArray failed**

**Explanation:** JVM_GetClassContext failed to get storage by way of stAllocArray.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java -Xmx option.

**JVMCI017 OutOfMemoryError, can't allocate new object**

**Explanation:** JVM_AllocateNewObject failed to get storage by way of stAllocArray.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java -Xmx option.

**JVMCI018 OutOfMemoryError, can't allocate new array**

**Explanation:** JVM_AllocateNewArray failed to get storage by way of stAllocArray.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java -Xmx option.

**JVMCI019 OutOfMemoryError, can't allocate object**

**Explanation:** JVM_NewInstanceFromConstructor failed to get storage by way of stAllocArray.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java -Xmx option.

**JVMCI020 OutOfMemoryError, stInternString failed**

**Explanation:** JVM_InternString failed to either locate a matching string, or add a new string.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java -Xmx option.

**JVMCI021 OutOfMemoryError, translating exception message**

**Explanation:** Either not enough Java Stack is available, or cannot find C-to-Java-string converter NewStringJVMPlatform.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Specify a larger Java stack size when you start the JVM; for example, by using the Java -Xoss option.

**JVMCI022 Cannot allocate memory in jvmpi_calloc**

**Explanation:** Profiler Interface could not get enough storage.

**System action:** The JVM prints the message "**Out of Memory, aborting**" and terminates abnormally.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

**JVMCI023 Cannot allocate memory to collect heap dump in jvmpi_heap_dump**

**Explanation:** Profiler Interface could not get enough storage.

**System action:** The JVM prints the message ″**Out of Memory, aborting**″ and terminates abnormally.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

**JVMCI024 Cannot allocate memory to collect heap dump in jvmpi_monitor_dump**

**Explanation:** Profiler Interface could not get enough storage.

**System action:** The JVM prints the message .**Out of Memory, aborting**. and terminates abnormally.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

**JVMCI025 Unable to open options file %filename**

**Explanation:** JNI_CreateJVM initialization detected the -Xoptionsfile option, but could not open the specified file.

**System action:** JNI_CreateJVM returns -1, to indicate that initialization of the JVM failed.

**User response:** Ensure that -Xoptionsfile specifies a valid file that the JVM can read.

**JVMCI026 Unable to determine the size of the options file %filename**

**Explanation:** JNI_CreateJVM initialization detected the -Xoptionsfile option, but could not obtain size information about the specified file by using fseek(fd,0L,SEEK_END) and ftell(fd).

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Ensure that -Xoptionsfile specifies a valid file that is not a socket or PIPE.

**JVMCI027 Unable to obtain memory to process %filename**

**Explanation:** JNI_CreateJVM initialization detected the -Xoptionsfile option, but could not allocate memory that is equal to the size of the specified file.

**System action:** JNI_CreateJVM returns -4 to indicate that initialization of the JVM failed.

**User response:** Check the size of the file that -Xoptionsfile specifies. The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

**JVMCI028 Error reading options file: %filename fread() returns %filesize: %strerror(errno)**

**Explanation:** JNI_CreateJVM initialization detected the -Xoptionsfile option, but could not read the specified file by using fread().

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Investigate the description for the given errno, and ensure that -Xoptionsfile specifies a valid file that is not a socket or PIPE.

**JVMCI029 Unable to obtain memory**

**Explanation:** JNI_CreateJVM initialization detected the -Xoptionsfile option, but could not allocate memory that is equal to the size of the specified file plus the string ″ -Xoptionsfile=″.

**System action:** JNI_CreateJVM returns -4 to indicate that initialization of the JVM failed.

**User response:** Check the size of the file that -Xoptionsfile specified. The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

**JVMCI030 Bad entry in options file: %filename**

**Explanation:** JNI_CreateJVM initialization parsed the contents of the file that the -Xoptionsfile option specified, and expected to find an option string beginning with the character ″-″.

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Check the contents of the file that -Xoptionsfile specified.

**JVMCI031 Bad entry in options file: %filename**

**Explanation:** JNI_CreateJVM initialization parsed the contents of the file that the -Xoptionsfile option specified, and expected to find an option string beginning with the character ″-″.

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Check the contents of the file that -Xoptionsfile specified.

**JVMCI032 Error parsing system properties within options file - rc=%rc**

**Explanation:** JNI_CreateJVM initialization parsed the contents of the file that the -Xoptionsfile option

specified, and found a problem while parsing a system property.

**System action:** JNI_CreateJVM returns a negative number to indicate that initialization of the JVM failed.

**User response:** Check the contents of the file that -Xoptionsfile specified.

---

**JVMCI033 Error parsing java options within options file - rc=%rc**

**Explanation:** JNI_CreateJVM initialization parsed the contents of the file that the -Xoptionsfile option specified, and found a problem while parsing a Java option setting.

**System action:** JNI_CreateJVM returns a negative number to indicate that initialization of the JVM failed.

**User response:** Check the contents of the file that -Xoptionsfile specified.

---

**JVMCI034 Cannot allocate memory during JVM initialization**

**Explanation:** JVM_StartThread attempted to create a new thread before JNI_CreateJVM was complete.

**System action:** The JVM prints the message "**Out of Memory, aborting**" and terminates abnormally.

**User response:** Determine why Thread.start() has been invoked before initialization of the JVM is complete.

---

**JVMCI035 Cannot override bootclasspath in Worker JVM**

**Explanation:** In a shared class environment, JNI_CreateJVM initialization parsed the properties that were specified for a Worker JVM, and found a setting for either "java.endorsed.dirs" or "ibm.jvm.bootclasspath".

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Check the properties that are specified for Worker JVMs.

---

**JVMCI037 Cannot use debugger (-Xdebug) with shared classes (-Xjvmset)**

**Explanation:** In a shared class environment, JNI_CreateJVM initialization does not allow -Xdebug to be specified.

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Do ***not*** specify the java -Xdebug option in conjunction with -Xjvmset.

---

**JVMCI038 Out of Shared Memory on property storage allocation**

**Explanation:** In a shared class environment, JNI_CreateJVM initialization failed to allocate shared memory for the purpose of copying shared system properties across the JVMSet.

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Refer to the shared classes *User Guide* for options that control shared memory.

---

**JVMCI039 Out of Shared Memory on property storage allocation**

**Explanation:** In a shared class environment, JNI_CreateJVM initialization failed to allocate shared memory for the purpose of copying shared system properties across the JVMSet.

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Refer to the shared classes *User Guide* for options that control shared memory.

---

**JVMCI040 Cannot configure system property %sharedpropertyname in Worker JVM**

**Explanation:** In a shared class environment, JNI_CreateJVM initialization found, for the specified property, an existing entry that is not allowed to be overwritten.

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Remove duplicate references to the specified system property.

---

**JVMCI041 unsafe get/set**

**Explanation:** A trusted system class has invoked Unsafe.getObject() with a NULL object.

**System action:** The JVM throws a NullPointerException.

**User response:** Contact your IBM service representative.

---

**JVMCI042 unsafe get/set**

**Explanation:** A trusted system class has invoked Unsafe.putObject() with a NULL object.

**System action:** The JVM throws a NullPointerException.

**User response:** Contact your IBM service representative.

---

**JVMCI043 unsafe get/set**

**Explanation:** A trusted system class has invoked Unsafe.get##type() with a NULL object. ##type## can be any of: Boolean, Byte, Short, Char, Int, or Float.

**System action:** The JVM throws a NullPointerException.

**User response:** Contact your IBM service representative.

**JVMCI044 unsafe get/set**

**Explanation:** A trusted system class has invoked Unsafe.put##type() with a NULL object. ##type## can be any of: Boolean, Byte, Short, Char, Int, or Float.

**System action:** The JVM throws a NullPointerException.

**User response:** Contact your IBM service representative.

**JVMCI045 Illegal size passed to allocateMemory**

**Explanation:** A trusted system class has invoked Unsafe.allocateMemory() with a negative size.

**System action:** The JVM throws an IllegalArgumentException.

**User response:** Contact your IBM service representative.

**JVMCI046 allocateMemory failed**

**Explanation:** A trusted system class has invoked Unsafe.allocateMemory(), but it could not get enough storage.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

**JVMCI047 Illegal size passed to reallocateMemory**

**Explanation:** A trusted system class has invoked Unsafe.reallocateMemory() with a negative size.

**System action:** The JVM throws an IllegalArgumentException.

**User response:** Contact your IBM service representative.

**JVMCI048 reallocateMemory failed**

**Explanation:** A trusted system class has invoked Unsafe.reallocateMemory(), but it could not get enough storage.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

**JVMCI049 Illegal size passed to copyMemory**

**Explanation:** A trusted system class has invoked Unsafe.copyMemory() with a negative size.

**System action:** The JVM throws an IllegalArgumentException.

**User response:** Contact your IBM service representative.

**JVMCI050 Illegal size passed to setMemory**

**Explanation:** A trusted system class has invoked Unsafe.setMemory() with a negative size.

**System action:** The JVM throws an IllegalArgumentException.

**User response:** Contact your IBM service representative.

**JVMCI051 Null field passed to staticFieldOffset**

**Explanation:** A trusted system class has invoked Unsafe.staticFieldOffset() with a NULL field object.

**System action:** The JVM throws a NullPointerException.

**User response:** Contact your IBM service representative.

**JVMCI052 defineClass**

**Explanation:** A trusted system class has invoked Unsafe.defineClass() with a NULL byte array.

**System action:** The JVM throws a NullPointerException.

**User response:** Contact your IBM service representative.

**JVMCI053 defineClass**

**Explanation:** A trusted system class has invoked Unsafe.defineClass() with a with a negative length argument.

**System action:** The JVM throws an ArrayIndexOutOfBoundsException.

**User response:** Contact your IBM service representative.

**JVMCI054 defineClass**

**Explanation:** A trusted system class has invoked Unsafe.defineClass(), but could not get enough storage.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

**JVMCI055 Cannot allocate assertion directives**

**Explanation:** Cannot allocate space for an instance of class AssertionStatusDirectives.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java -Xmx option.

**JVMCI056 Cannot allocate assertion directives**

**Explanation:** Cannot allocate space for an instance of class AssertionStatusDirectives.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java -Xmx option.

**JVMCI057 Cannot allocate assertion directives**

**Explanation:** Cannot allocate space for an instance of class AssertionStatusDirectives.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java -Xmx option.

**JVMCI058 Unsafe_StaticFieldBase**

**Explanation:** A trusted system class has invoked Unsafe.staticFieldBase() with a NULL field object.

**System action:** The JVM throws a NullPointerException.

**User response:** Contact your IBM service representative.

**JVMCI059 Unsafe_EnsureClassInitialized**

**Explanation:** A trusted system class has invoked Unsafe.ensureClassInitialized() with a NULL class object.

**System action:** The JVM throws a NullPointerException.

**User response:** Contact your IBM service representative.

**JVMCI060 Unsafe_ArrayBaseOffSet**

**Explanation:** A trusted system class has invoked Unsafe.arrayBaseOffSet() with a non-array class.

**System action:** The JVM throws an InvalidClassException.

**User response:** Contact your IBM service representative.

**JVMCI061 Unsafe_ArrayIndexScale**

**Explanation:** A trusted system class has invoked Unsafe.arrayIndexScale() with a non-array class.

**System action:** The JVM throws an InvalidClassException.

**User response:** Contact your IBM service representative.

**JVMCI062 holdsLock**

**Explanation:** Thread.holdsLock() has been invoked with a NULL object argument.

**System action:** The JVM throws a NullPointerException.

**User response:** Examine invocations of Thread.holdsLock(). If you cannot solve the problem, contact your IBM service representative.

**JVMCI063 OutOfMemoryError, GetStringChars failed**

**Explanation:** A call to jni_getStringChars() has failed because not enough memory is available.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

**JVMCI064 unsafe getLong**

**Explanation:** A trusted system class has invoked Unsafe.getLong() with a NULL object.

**System action:** The JVM throws a NullPointerException.

**User response:** Contact your IBM service representative.

**JVMCI065 unsafe putLong**

**Explanation:** A trusted system class has invoked Unsafe.putLong() with a NULL object.

**System action:** The JVM throws a NullPointerException.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

**JVMCI066 unsafe getDouble**

**Explanation:** A trusted system class has invoked Unsafe.getDouble() with a NULL object.

**System action:** The JVM throws a NullPointerException.

**User response:** Contact your IBM service representative.

**JVMCI067 unsafe putDouble**

**Explanation:** A trusted system class has invoked Unsafe.putDouble() with a NULL object.

**System action:** The JVM throws a NullPointerException.

**User response:** Contact your IBM service representative.

**JVMCI068 Cannot set system assertion status in Worker JVM**

**Explanation:** In a shared class environment, JNI_CreateJVM initialization parsed the properties that were specified for a Worker JVM, and found either -enablesystemassertions or -disablesystemassertions. Worker JVMs are not permitted to set these options.

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Ensure that the properties that are specified for Worker JVMs do ***not*** include Java assert options. For help on assert options, run `java -assert`.

**JVMCI069 Cannot set shared class maximum option -Xscmax in a Worker JVM**

**Explanation:** In a shared class environment, JNI_CreateJVM initialization parsed the properties that were specified for a Worker JVM, and found -Xscmax. Worker JVMs are not permitted to set this option.

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Ensure that the properties that are specified for Worker JVMs do ***not*** include the Java option -Xscmax.

**JVMCI070 JVMMI object enumeration - invalid class found at 0x%objectpointer**

**Explanation:** The monitoring interface was invoked to list objects in the heap, but could not obtain class information for an object.

**System action:** The callback function for this object is not called and no further information is produced for this object. Enumeration continues with the next object in the heap.

**User response:** Contact your IBM service representative.

**JVMCI071 JVMMI object enumeration - object at 0x%objectpointer has null class block**

**Explanation:** The monitoring interface was invoked to list objects in the heap, but found a NULL class block for an object.

**System action:** The callback function for this object is not called and no further information is produced for this object. Enumeration continues with the next object in the heap.

**User response:** Contact your IBM service representative.

**JVMCI072 JVMMI object enumeration - unrecognized array object at 0x%objectpointer**

**Explanation:** The monitoring interface was invoked to list objects in the heap, and determined that an object was a multidimensional primitive array, but could not determine the type of one of the dimensions.

**System action:** The callback function for this object is not called, and enumeration continues.

**User response:** Contact your IBM service representative.

**JVMCI073 JVMMI object enumeration - unrecognized primitive array at 0x%objectpointer**

**Explanation:** The monitoring interface was invoked to list objects in the heap, and determined that an object was an array, but could not determine the type of array.

**System action:** The callback function for this object is not called, and enumeration continues.

**User response:** Contact your IBM service representative.

**JVMCI074 Cannot allocate memory in jvmpi_interface**

**Explanation:** The monitoring interface could not get enough storage.

**System action:** The JVM prints the message "**Out of Memory, aborting**" and terminates abnormally.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

**JVMCI075 Cannot allocate memory in jvmpi_dump_object_event**

**Explanation:** The monitoring interface could not get enough storage.

**System action:** The JVM prints the message "**Out of Memory, aborting**" and terminates abnormally.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

**JVMCI076 JVM is requesting a heap dump**

**Explanation:** The monitoring interface was invoked to generate a dump of the heap.

**System action:** A heap dump is requested.

**User response:** None.

**JVMCI077 Heap dump complete**

**Explanation:** The monitoring interface has generated a dump of the heap.

**System action:** A heap dump has been generated.

**User response:** None.

**JVMCI078 NullPointerException, mangleMethodName passed NULL MethodBlock**

**Explanation:** During the invocation of a native method, a NULL method block was found.

**System action:** The JVM throws a NullPointerException.

**User response:** Contact your IBM service representative.

**JVMCI079 NullPointerException, maxMangledMethodNameLength passed NULL MethodBlock**

**Explanation:** During the invocation of a native method, a NULL method block was found.

**System action:** The JVM throws a NullPointerException.

**User response:** Contact your IBM service representative.

**JVMCI080 SetMirroredProtectionDomains NULL cb**

**Explanation:** SecureClassLoader.setMirroredProtectionDomain() was invoked with a NULL class.

**System action:** The JVM throws a NullPointerException.

**User response:** Contact your IBM service representative.

**JVMCI081 Clone NULL this pointer**

**Explanation:** Object.clone() "this" instance is NULL.

**System action:** The JVM throws a NullPointerException.

**User response:** Contact your IBM service representative.

**JVMCI082 GetClassLoader NULL cb**

**Explanation:** Class.getClassLoader() "this" instance is NULL.

**System action:** The JVM throws a NullPointerException.

**User response:** Contact your IBM service representative.

**JVMCI083 IsInterface NULL cb**

**Explanation:** Class.isInterface() "this" instance is NULL.

**System action:** The JVM throws a NullPointerException.

**User response:** Contact your IBM service representative.

**JVMCI084 GetClassSigners NULL cb**

**Explanation:** Class.getClassSigners() "this" instance is NULL.

**System action:** The JVM throws a NullPointerException.

**User response:** Contact your IBM service representative.

**JVMCI085 SetClassSigners NULL cb**

**Explanation:** Class.setClassSigners() "this" instance is NULL.

**System action:** The JVM throws a NullPointerException.

**User response:** Contact your IBM service representative.

---

**JVMCI086 GetProtectionDomain NULL cb**

**Explanation:** Class.getProtectionDomain() ″this″ instance is NULL.

**System action:** The JVM throws a NullPointerException.

**User response:** Contact your IBM service representative.

---

**JVMCI087 SetProtectionDomain NULL cb**

**Explanation:** Class.setProtectionDomain() ″this″ instance is NULL.

**System action:** The JVM throws a NullPointerException.

**User response:** Contact your IBM service representative.

---

**JVMCI088 OutOfMemoryError, _CharToByteLength cannot obtain TLS**

**Explanation:** A call to _CharToByteLength() has failed because it could not get thread local storage.

**System action:** The JVM throws an OutOfMemoryError. _CharToByteLength() returns -2.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

---

**JVMCI089 OutOfMemoryError, _CharToByteLength cannot allocate cache**

**Explanation:** A call to _CharToByteLength() has failed because it could not allocate storage.

**System action:** The JVM throws an OutOfMemoryError. _CharToByteLength() returns -2.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

---

**JVMCI090 OutOfMemoryError, _CharToByteLength cannot allocate buffer**

**Explanation:** A call to _CharToByteLength() has failed because it could not allocate storage.

**System action:** The JVM throws an OutOfMemoryError. _CharToByteLength() returns -2.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

---

**JVMCI091 Null field passed to objectFieldOffset**

**Explanation:** Unsafe.ObjectFieldOffset() was invoked with a NULL field.

**System action:** The JVM throws a NullPointerException.

**User response:** Contact your IBM service representative.

---

**JVMCI092 Out of Shared Memory on property storage allocation.**

**Explanation:** In a shared class environment, JNI_CreateJVM initialization failed to allocate shared memory for the copying of shared system properties across the JVMSet.

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Refer to the shared classes *User Guide* for options that control shared memory.

---

**JVMCI093 Unable to load Core Interface - JVM Anchor Reference is missing**

**Explanation:** JNI_CreateJVM initialization found a null pointer to the JVM Anchor block.

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Contact your IBM service representative.

---

**JVMCI094 Unable to load Core Interface - JVM initialization argument is missing**

**Explanation:** JNI_CreateJVM initialization found a null pointer to the JVM initialization arguments.

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Contact your IBM service representative.

---

**JVMCI095 JavaVM Init Args is not present, jvm pointer = %p**

**Explanation:** JNI_CreateJVM initialization found a null pointer to the JVM initialization arguments.

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Contact your IBM service representative.

---

**JVMCI096 Unable to load HPI - JVM initialization arguments missing**

**Explanation:** JNI_CreateJVM initialization found a null pointer to the JVM initialization arguments.

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Contact your IBM service representative.

**JVMCI097 JVM Instance is not present**

**Explanation:** JNI_CreateJVM initialization found a null pointer to the JVM Anchor block.

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Contact your IBM service representative.

**JVMCI098 xmloadJVMHelperLib %s %s, failed**

**Explanation:** JNI_CreateJVM initialization could not load the indicated shared library with its associated options. Message JVMCI158 might also be printed.

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Contact your IBM service representative.

**JVMCI099 Unable to trace user arguments - JVM Instance is not present**

**Explanation:** JNI_CreateJVM initialization found a null pointer to the JVM Anchor block.

**System action:** JNI_CreateJVM continues, but without tracing user arguments.

**User response:** Contact your IBM service representative.

**JVMCI100 Unable to trace user arguments - no arguments supplied, jvm pointer = %p**

**Explanation:** JNI_CreateJVM initialization found a null pointer to the JVM initialization arguments.

**System action:** JNI_CreateJVM continues, but without tracing user arguments.

**User response:** Contact your IBM service representative.

**JVMCI101 Property Table is not present**

**Explanation:** JNI_CreateJVM initialization found a null pointer to the JVM property table.

**System action:** JNI_CreateJVM returns -4 to indicate that initialization of the JVM failed.

**User response:** Contact your IBM service representative.

**JVMCI102 Out of Memory on property storage allocation**

**Explanation:** JNI_CreateJVM initialization could not allocate storage for a property table entry.

**System action:** JNI_CreateJVM returns -4 to indicate that initialization of the JVM failed.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

**JVMCI103 Out of Memory on property add**

**Explanation:** JNI_CreateJVM initialization could not allocate storage for a property table entry name or value field.

**System action:** JNI_CreateJVM returns -4 to indicate that initialization of the JVM failed.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

**JVMCI104 Failed to locate entry point %s**

**Explanation:** JNI_CreateJVM initialization could not locate an entry point named ciInit in the Core Interface Library that was specified with the -Dibm.load.ci= argument.

**System action:** JNI_CreateJVM continues, but with the default core interface.

**User response:** Check whether the library that was specified by -Dibm.load.ci= contains an entry point for ciInit.

**JVMCI106 Failed to find library %s**

**Explanation:** JNI_CreateJVM initialization could not locate the core interface library that was specified with the -Dibm.load.ci= argument.

**System action:** JNI_CreateJVM continues, but with the default core interface.

**User response:** Check whether the library that was specified by -Dibm.load.ci= is accessible to the JVM.

**JVMCI107 Unable to allocate memory for Library Name Property**

**Explanation:** JNI_CreateJVM initialization could not allocate storage to store the string ibm.load.XX.nt.

**System action:** JNI_CreateJVM returns -4 to indicate that initialization of the JVM failed.

**User response:** The C-runtime heap of the process

(not the Java object heap) is full. Increase the heap if that is possible in your environment.

---

**JVMCI108 Unable to allocate memory for Initial Function Name**

**Explanation:** JNI_CreateJVM initialization could not allocate storage to store the string xxInit.

**System action:** JNI_CreateJVM returns -4 to indicate that initialization of the JVM failed.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

---

**JVMCI109 No initialization point found for sub component %s**

**Explanation:** JNI_CreateJVM initialization could not locate a default initialization entry point of the form xxInit for the indicated sub component xx.

**System action:** JNI_CreateJVM attempts to continue.

**User response:** Contact your IBM service representative.

---

**JVMCI110 ciFacade is not present**

**Explanation:** JNI_CreateJVM initialization found a null pointer to the JVM core interface facade.

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Contact your IBM service representative.

---

**JVMCI111 ciFacade version is not supported %d**

**Explanation:** JNI_CreateJVM initialization found an nonvalid version number as indicated.

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Contact your IBM service representative.

---

**JVMCI112 Unrecognized JNI version: 0x%08x**

**Explanation:** JNI_CreateJVM initialization found an nonvalid version number as indicated.

**System action:** JNI_CreateJVM returns -3 to indicate that initialization of the JVM failed.

**User response:** Contact your IBM service representative.

---

**JVMCI113 Cannot obtain iconv converters**

**Explanation:** JNI_CreateJVM initialization could not initialize iconv converters.

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Contact your IBM service representative.

---

**JVMCI114 Cannot obtain system-specific information**

**Explanation:** JNI_CreateJVM initialization could not obtain system properties (home directory/DLL directory/system classpath, calculated from the location of libjava.so).

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Contact your IBM service representative.

---

**JVMCI115 IBM_JAVA_OPTIONS error**

**Explanation:** JNI_CreateJVM initialization could not obtain storage to store the value of the IBM_JAVA_OPTIONS environment variable.

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

---

**JVMCI116 Error setting JVM default options - rc=%d**

**Explanation:** JNI_CreateJVM initialization detected an error while parsing JVM options. This message might be preceded by other, more-specific messages.

**System action:** JNI_CreateJVM returns the code that is shown in the message, to indicate that initialization of the JVM failed.

**User response:** Check the options that are passed to the JVM. If you cannot solve the problem, contact your IBM service representative.

---

**JVMCI117 Bad IBM_JAVA_OPTIONS: %s**

**Explanation:** JNI_CreateJVM initialization expects options to begin with a "-" (hyphen).

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Check the options that are passed to the JVM. If you cannot solve the problem, contact your IBM service representative.

**JVMCI118 Error parsing IBM_JAVA_OPTIONS system properties - rc=%d**

**Explanation:** JNI_CreateJVM initialization detected an error while parsing JVM options. This message might be preceded by other, more-specific messages.

**System action:** JNI_CreateJVM returns the code that is shown in the message, to indicate that initialization of the JVM failed.

**User response:** Check the options that are passed to the JVM. If you cannot solve the problem, contact your IBM service representative.

**JVMCI119 Error parsing IBM_JAVA_OPTIONS java options - rc=%d**

**Explanation:** JNI_CreateJVM initialization detected an error while parsing JVM options. This message might be preceded by other, more-specific messages.

**System action:** JNI_CreateJVM returns the code that is shown in the message, to indicate that initialization of the JVM failed.

**User response:** Check the options that are passed to the JVM. If you cannot solve the problem, contact your IBM service representative.

**JVMCI120 Unable to parse System Properties - no argument supplied, jvm pointer = %p**

**Explanation:** JNI_CreateJVM initialization found a null pointer to the JVM initialization arguments.

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Contact your IBM service representative.

**JVMCI121 Unable to parse 1.2 format System Properties - rc=%d**

**Explanation:** JNI_CreateJVM initialization detected an error while parsing JVM options. This message might be preceded by other, more-specific messages.

**System action:** JNI_CreateJVM returns the code that is shown in the message, to indicate that initialization of the JVM failed.

**User response:** Check the options that are passed to the JVM. If you cannot solve the problem, contact your IBM service representative.

**JVMCI122 Unable to parse supplied options - no argument supplied, jvm pointer = %p**

**Explanation:** JNI_CreateJVM initialization found a null pointer to the JVM initialization arguments.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Contact your IBM service representative.

**JVMCI123 Unable to parse 1.2 format supplied options - rc=%d**

**Explanation:** JNI_CreateJVM initialization detected an error while parsing JVM options. This message might be preceded by other, more-specific messages.

**System action:** JNI_CreateJVM returns the code that is shown in the message, to indicate that initialization of the JVM failed.

**User response:** Check the options that are passed to the JVM. If you cannot solve the problem, contact your IBM service representative.

**JVMCI124 Bad Service Option: %s**

**Explanation:** JNI_CreateJVM initialization expects individual options to begin with a "-" (hyphen).

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xservice=** options that are indicated in the message. Correct as necessary.

**JVMCI125 Bad Service Option: %s**

**Explanation:** JNI_CreateJVM initialization expects individual options to begin with a "-" (hyphen).

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xservice=** options that are indicated in the message. Correct as necessary.

**JVMCI126 Error parsing Service system properties - rc=%d**

**Explanation:** JNI_CreateJVM initialization detected an error while parsing JVM options. This message might be preceded by other, more-specific messages.

**System action:** JNI_CreateJVM returns the code that is shown in the message, to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xservice=** options that are indicated in the message. Correct as necessary.

**JVMCI127 Error parsing Service java options - rc=%d**

**Explanation:** JNI_CreateJVM initialization detected an error while parsing JVM options. This message might be preceded by other, more-specific messages.

**System action:** JNI_CreateJVM returns the code that is shown in the message, to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xservice=** options that are indicated in the message. Correct as necessary.

---

**JVMCI128 Illegal option: %s**

**Explanation:** JNI_CreateJVM initialization detected a **-verbose** option that had no following "." (period) character. The string that follows -verbose is shown in the message.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-verbose:** options that are passed to the JVM. Correct as necessary.

---

**JVMCI129 Unrecognized verbose option: %s**

**Explanation:** JNI_CreateJVM initialization detected a **-verbose** option that had no valid option following the "." (period) character. The string that follows -verbose is shown in the message

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-verbose** options that are passed to the JVM. Correct as necessary. You can list valid options by running `java -?`.

---

**JVMCI130 Illegal option: %s**

**Explanation:** JNI_CreateJVM initialization detected an **-verify** option that had no valid option following. The string following -verify is shown in the message.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-verify** option that is passed to the JVM. Correct as necessary. You can list valid options by running `java -?`.

---

**JVMCI131 Invalid number of threads: %s**

**Explanation:** JNI_CreateJVM initialization detected an **-Xgcthreads** option that had no positive integer following. The string that follows -Xgcthreads is shown in the message.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xgcthreads** option that is passed to the JVM. Correct as necessary.

---

**JVMCI133 Bad concurrent GC level: %s**

**Explanation:** JNI_CreateJVM initialization detected an **-Xconcurrentlevel** option that had no valid size following. You can specify sizes as a positive integers that are optionally suffixed with K, M or G to indicate units of KB, MB, or GB. The string that follows -Xconcurrentlevel is shown in the message.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xconcurrentlevel** option that is passed to the JVM. Correct as necessary.

---

**JVMCI134 Incorrect usage of -Xverbosegclog**

**Explanation:** JNI_CreateJVM initialization detected an **-Xverbosegclog** option that had no valid suboptions following. The string that follows -Xverbosegclog is shown in the message.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xverbosegclog** option that is passed to the JVM. Correct as necessary.

---

**JVMCI135 Illegal option: %s**

**Explanation:** JNI_CreateJVM initialization detected an **-Xgcpolicy** option that had no suboptions following. The string that follows -Xgcpolicy is shown in the message.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xgcpolicy** option that is passed to the JVM. Correct as necessary. You can list valid options by running `java -?`.

---

**JVMCI136 Illegal option: %s**

**Explanation:** JNI_CreateJVM initialization detected an **-Xgcpolicy** option that had no valid suboption following. The string that follows -Xgcpolicy is shown in the message.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xgcpolicy** option that is passed to the JVM. Correct as necessary. You can list valid options by running `java -?`.

---

**JVMCI137 Bad native stack size: %s**

**Explanation:** JNI_CreateJVM initialization detected an **-Xss** option that had no valid size following. You can specify sizes as a positive integers that are optionally suffixed with K, M or G to indicate units of KB, MB, or GB. Size must be equivalent to, or greater than, 1000 bytes. The string that follows -Xss is shown in the message.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xss** option that is passed to the JVM. Correct as necessary. You can list valid options by running `java -?`.

---

**JVMCI138 Bad Java stack size: %s**

**Explanation:** JNI_CreateJVM initialization detected an **-Xoss** option that had no valid size following. You can specify sizes as a positive integers that are optionally suffixed with K, M or G to indicate units of KB, MB, or GB. Size must be equivalent to, or greater than, 1000 bytes. The string that follows -Xoss is shown in the message.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xoss** option that is passed to the JVM. Correct as necessary. You can list valid options by running `java -?`.

**JVMCI139 Bad init heap size: %s**

**Explanation:** JNI_CreateJVM initialization detected an **-Xms** option that had no valid size following. You can specify sizes as a positive integers that are optionally suffixed with K, M or G to indicate units of KB, MB, or GB. The string that follows -Xms is shown in the message.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xms** option that is passed to the JVM. Correct as necessary. You can list valid options by running `java -?`.

**JVMCI140 Bad max heap size: %s**

**Explanation:** JNI_CreateJVM initialization detected an **-Xmx** option that had no valid size following. You can specify sizes as a positive integers that are optionally suffixed with K, M or G to indicate units of KB, MB, or GB. The string that follows -Xmx is shown in the message.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xmx** option that is passed to the JVM. Correct as necessary. You can list valid options by running `java -?`.

**JVMCI141 Bad maxHeapFreePercent size: %s**

**Explanation:** JNI_CreateJVM initialization detected an **-Xmaxf** option that had no valid value following. Valid values are 0 and 1.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xmaxf** option that is passed to the JVM. Correct as necessary.

**JVMCI142 Bad minHeapFreePercent size: %s**

**Explanation:** JNI_CreateJVM initialization detected an **-Xminf** option that had no valid value following. Valid values are 0 and 1.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xminf** option that is passed to the JVM. Correct as necessary.

**JVMCI143 Bad maxHeapExpansion size: %s**

**Explanation:** JNI_CreateJVM initialization detected an **-Xmaxe** option that had no valid size following. You can specify sizes as a positive integers that are optionally suffixed with K, M or G to indicate units of KB, MB, or GB. The string that follows -Xmaxe is shown in the message.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xmaxe** option that is passed to the JVM. Correct as necessary.

**JVMCI144 Bad minHeapExpansion size: %s**

**Explanation:** JNI_CreateJVM initialization detected an **-Xmine** option that had no valid size following. You can specify sizes as a positive integers that are optionally suffixed with K, M or G to indicate units of KB, MB, or GB. The string that follows -Xmine is shown in the message.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xmine** option that is passed to the JVM. Correct as necessary.

**JVMCI146 Bad initial Transient heap size: %s**

**Explanation:** In a shared class environment, JNI_CreateJVM initialization detected an **-Xinitth** option that had no valid size following. You can specify sizes as a positive integers that are optionally suffixed with K, M or G to indicate units of KB, MB, or GB. The string that follows -Xinitth is shown in the message.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xinitth** option that is passed to the JVM. Correct as necessary. You can list valid options by running `java -?`.

**JVMCI147 Bad initial System heap size: %s**

**Explanation:** In a shared class environment, JNI_CreateJVM initialization detected an **-Xinitsh** option that had no valid size following. You can specify sizes as a positive integers that are optionally suffixed

with K, M or G to indicate units of KB, MB, or GB. The string that follows -Xinitsh is shown in the message.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xinitsh** option that is passed to the JVM. Correct as necessary. You can list valid options by running `java -?`.

---

**JVMCI148 Bad initial ACS heap size: %s**

**Explanation:** In a shared class environment, JNI_CreateJVM initialization detected an **-Xinitacsh** option that had no valid size following. You can specify sizes as a positive integers that are optionally suffixed with K, M or G to indicate units of KB, MB, or GB. The string that follows -Xinitacsh is shown in the message.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xinitacsh** option that is passed to the JVM. Correct as necessary. You can list valid options by running `java -?`.

---

**JVMCI149 Bad shared memory size: %s**

**Explanation:** In a shared class environment, JNI_CreateJVM initialization detected a Master JVM **-Xjvmset** option that had no valid size following. You can specify sizes as a positive integers that are optionally suffixed with K, M or G to indicate units of KB, MB, or GB. The string that follows -Xjvmset is shown in the message.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xjvmset** option that is passed to the JVM. Correct as necessary. You can list valid options by running `java -?`.

---

**JVMCI150 Invalid use of %s option**

**Explanation:** In a shared class environment, JNI_CreateJVM initialization detected a Worker JVM **-Xjvmset** option that had suboptions following. You must not specify suboptions for a Worker JVM. The string that follows -Xjvmset is shown in the message.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xjvmset** option that is passed to the JVM. Correct as necessary. You can list valid options by running `java -?`.

---

**JVMCI151 Invalid shared class option setting: %s**

**Explanation:** JNI_CreateJVM initialization detected an **-Xscmax** option that had no valid size following. Size must be a positive integer 2048 through 1048576. The string that follows -Xscmax is shown in the message.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xscmax** option that is passed to the JVM. Correct as necessary.

---

**JVMCI152 Invalid option : %s**

**Explanation:** JNI_CreateJVM initialization detected one of the **-ea**, **-enableassertions**, **-da**, or **-disableassertions** options that had no following "." (period) character.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the assert options that are passed to the JVM. Correct as necessary. You can list valid options by running `java -assert`.

---

**JVMCI153 Invalid option, optionString pointer is null**

**Explanation:** JNI_CreateJVM initialization detected an option that started with -X, but had no further value.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the assert options that are passed to the JVM. Correct as necessary. You can list valid options by running `java -assert`.

---

**JVMCI155 Specified options prevent use of JIT**

**Explanation:** JNI_CreateJVM initialization detected either the **-Xt** or the **-Xmt** option, and has disabled the JIT compiler..

**System action:** JNI_CreateJVM continues with the JIT compiler disabled.

**User response:** None.

---

**JVMCI156 OutOfMemoryError, IBMJVM_ResizeArray - stAllocArray for new array failed**

**Explanation:** A call to ExtendedSystem.resizeArray() has failed because not enough memory is available.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java -Xmx option.

---

**JVMCI157 OutOfMemoryError, stClonePrimitiveArrayToSystemHeap for GetSystemHeapArray() failed**

**Explanation:** A call to ClassLoader.getSystemHeapArray() has failed because not enough memory is available.

**System action:** The JVM throws an OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java -Xmx option.

---

**JVMCI158 Can't load %s, because %s**

**Explanation:** JNI_CreateJVM initialization could not load the indicated shared library because of the reason indicated. Message JVMCI098 might also be printed.

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Contact your IBM service representative.

---

**JVMCI159 Unable to initialize JVM helper library %s**

**Explanation:** JNI_CreateJVM initialization could not call the OnLoad() entry point for the indicated shared library.

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Contact your IBM service representative.

---

**JVMCI160 Corrupted JVM helper library %s**

**Explanation:** JNI_CreateJVM initialization could not call the OnLoad() entry point for the indicated shared library.

**System action:** JNI_CreateJVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Contact your IBM service representative.

---

**JVMCI161 FATAL ERROR in native method: %s**

**Explanation:** A JNI function has found an unrecoverable error. Information that is specific to the error is included in the message.

**System action:** The JVM prints a Java stack trace and ends. Dumps might be produced, depending upon the setting of the JAVA_DUMP_OPTS environment variable.

**User response:** Examine the message detail and Java stack trace to determine the possible cause of the error. If you cannot solve the problem, contact your IBM service representative.

---

**JVMCI162 Profiler error**

**Explanation:** JNI_CreateJavaVM has determined that the profiling interface is not initialized.

**System action:** JNI_CreateJavaVM returns -1 to indicate that initialization of the JVM failed.

**User response:** Contact your IBM service representative.

---

**JVMCI163 Illegal string length in JVMMI heapdump**

**Explanation:** A string is greater than the maximum JVMMI reference buffer size of 1024 bytes.

**System action:** The string is ignored and the JVM attempts to continue.

**User response:** Contact your IBM service representative.

---

**JVMCI164 Illegal option: %s**

**Explanation:** JNI_CreateJVM initialization detected a **-Xifa** option that had no valid value following. Valid values are on, off, force & project.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xifa** option that is passed to the JVM. Correct as necessary

---

**JVMCI165 Illegal option specified %s**

**Explanation:** JNI_CreateJVM initialization detected a **-Xk** option that had an invalid number of classes specified.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xk** option that is passed to the JVM. Correct as necessary.

---

**JVMCI166 Illegal option specified %s**

**Explanation:** JNI_CreateJVM initialization detected a **-Xp** option that had an invalid format for the following pCluster specification. The expected format is **-Xpiiii[K][,oooo[K]]**.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xp** option that is passed to the JVM. Correct as necessary.

---

**JVMCI167 Bad pCluster initial size %s**

**Explanation:** JNI_CreateJVM initialization detected a **-Xp** option that had an invalid initial pCluster size specified.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xp** option that is passed to the JVM. Correct as necessary.

**JVMCI168 Bad pCluster overflow size %s**

**Explanation:** JNI_CreateJVM initialization detected a **-Xp** option that had an invalid overflow pCluster size specified.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xp** option that is passed to the JVM. Correct as necessary.

**JVMCI169 Initial pCluster size less than pCluster overflow %s**

**Explanation:** JNI_CreateJVM initialization detected a **-Xp** option of the form **-Xpiiii[K][,oooo[K]]** where **iiii** is less than **oooo**.

**System action:** JNI_CreateJVM returns -6 to indicate that initialization of the JVM failed.

**User response:** Examine the **-Xp** option that is passed to the JVM. Correct as necessary.

# JVM error messages for JVMCL

**JVMCL001 OutOfMemoryError, dcUTF2JavaString failed**

**Explanation:** During the resolution of a Java constant pool string, the function that converts a UTF string type to an internal Java string type failed and returned a NULL value because of a lack of Java heap memory.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java -Mx option.

**JVMCL002 OutOfMemoryError, stInternString failed**

**Explanation:** During the resolution of a Java constant pool string, the function that resolves a Java string to a single and unique equivalent inside the JVM failed and returned a NULL value because of a lack of Java heap memory.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Use a larger maximum heap when you start the JVM; for example, by using the Java -Mx option.

**JVMCL003 OutOfMemoryError, unable to allocate storage for MethodTable**

**Explanation:** The system memory calloc (or shared memory alloced for a shared class in the sharable JVM) that was used to get storage for a method table has returned NULL.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. In a shared classes environment, see the shared classes *User Guide* for options that control shared memory.

**JVMCL004 OutOfMemoryError, unable to allocate storage for offset vector**

**Explanation:** While laying out the fields of a Java class in memory, storage is requested for the list of offsets of each field in the class. Not enough memory is available to honor this request.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. In a shared classes environment, see the shared classes *User Guide* for options that control shared memory, because the storage is managed by the JVM in this case.

**JVMCL005 OutOfMemoryError, unable to allocate storage for interface table**

**Explanation:** The system memory calloc (or shared memory allocated for a shared classes environment in the sharable JVM) that was used to get storage for a interface table has returned NULL.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment. In a shared classes environment, see the shared classes *User Guide* for options that control shared memory, because the storage is managed by the JVM in this case.

**JVMCL006 OutOfMemoryError, unable to allocate storage for MethodBlock**

**Explanation:** The system memory calloc (or shared memory alloced for a shared classes in the sharable JVM) that was used to get storage for a (miranda) method block has returned NULL.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment. In a shared classes environment, see the shared classes *User Guide* for options that control shared memory, because the storage is managed by the JVM in this case.

---

**JVMCL007 OutOfMemoryError, mbName returned NULL**

**Explanation:** The system memory alloc (or shared memory allocated for a shared classes in the sharable JVM) that was used to get storage for an entry in the cache of UTF8 strings has returned NULL during the preparation of a class-implemented method.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment. In a shared classes environment, see the shared classes *User Guide* for options that control shared memory, because the storage is managed by the JVM in this case.

---

**JVMCL008 OutOfMemoryError, stAddToLoadedClasses failed**

**Explanation:** The function that adds a class object to the internal list in the JVM has failed.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment. In a shared classes environment, see the shared classes *User Guide* for options that control shared memory, because the storage is managed by the JVM in this case.

---

**JVMCL009 OutOfMemoryError, sysMalloc for loading classes (file) failed**

**Explanation:** The malloc function that was used in class loading to create an internal buffer has returned NULL.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

---

**JVMCL010 OutOfMemoryError, sysMalloc for loading classes (zip) failed**

**Explanation:** The malloc function that was used in class loading to create an internal buffer has returned NULL.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

---

**JVMCL011 OutOfMemoryError, putPackage for loading classes (zip) failed**

**Explanation:** Adding a package table entry failed because of a lack of system memory (shared memory segment space in a shared classes environment).

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment. In a shared classes environment, the shared memory segment is full. See the sharable JVM documentation for information about how to increase the segment.

---

**JVMCL012 OutOfMemoryError, allocation failed**

**Explanation:** Class allocation has failed for an array or primitive class.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Start the JVM with a larger maximum heap; for example, by using the Java -Mx option. In a shared classes environment the storage area that is full depends on the type of class that is being allocated. See the sharable JVM documentation for further information.

---

**JVMCL013 OutOfMemoryError, unable to allocate storage for pool buffer**

**Explanation:** The calloc system function (or classSharedMalloc in a sharable JVM environment) has returned NULL.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment. In a shared classes environment, see the shared classes *User Guide* for options that control shared memory, because the storage is managed by the JVM in this case.

---

**JVMCL014 OutOfMemoryError, cbName returned NULL**

**Explanation:** Setting the class name of a class has failed, probably because of the inability to create a new UTF8 cache entry.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment. In a shared classes environment, see the shared classes *User Guide* for options that control shared memory, because the storage is managed by the JVM in this case.

**JVMCL015 OutOfMemoryError, allocation of class mirror failed**

**Explanation:** The function that was used to allocate an unmovable byte array in the JVM has returned NULL.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Use a larger maximum heap to start the JVM; for example, by using the java -Mx option. In a shared classes environment, the storage area that is full depends on the type of class that is being allocated. See the sharable JVM documentation for more information about how to increase the memory that is available for class storage.

**JVMCL016 OutOfMemoryError, jvmpi_load_class_hook returned NULL pointer**

**Explanation:** The JVMPI user-pluggable code has returned NULL in response to a class load event.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** If this is an unexpected result, investigate the code that uses the JVMPI interface to listen for the JVMPI CLASS_LOAD event.

**JVMCL017 OutOfMemoryError, loading classes**

**Explanation:** A generic out of memory error has occurred.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Use a larger maximum heap to start the JVM; for example, by using the Java -Mx option. In a shared classes environment, the storage area that is full depends on the type of class that is being allocated. See the sharable JVM documentation for more information about how to increase the memory that is available for class storage. If this does not solve the problem, increase the process C-heap that is allocated to the JVM process, if possible.

**JVMCL018 OutOfMemoryError, stAllocObject for new field instance failed**

**Explanation:** The Java heap is full.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Use a larger maximum heap to start the JVM; for example, by using the Java -Mx option. In a shared classes environment, the storage area that is full depends on the type of class that is being allocated. See the sharable JVM documentation for more information about how to increase the memory that is available for class storage.

**JVMCL019 OutOfMemoryError, stAllocArray for new array failed**

**Explanation:** The Java heap is full.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Use a larger maximum heap to start the JVM; for example, by using the Java -Mx option. In a shared classes environment, the storage area that is full depends on the type of class that is being allocate. See the sharable JVM documentation for more information about how to increase the memory that is available for class storage.

**JVMCL020 OutOfMemoryError, stAllocObject for new method failed**

**Explanation:** The Java heap is full.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Use a larger maximum heap to start the JVM; for example, by using the Java -Mx option. In a shared classes environment, the storage area that is full depends on the type of class that is being allocated. See the sharable JVM documentation for more information about how to increase the memory that is available for class storage.

**JVMCL021 OutOfMemoryError, stAllocObject for new constructor failed**

**Explanation:** The Java heap is full.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Use a larger maximum heap to start the JVM; for example, by using the Java -Mx option. In a shared classes environment, the storage area that is full depends on the type of class that is being allocated. See the sharable JVM documentation for more

information about how to increase the memory that is available for class storage.

**JVMCL022 OutOfMemoryError, sysMalloc for inner classes failed**

**Explanation:** The malloc system function has returned NULL.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Contact your IBM service representative.

**JVMCL023 OutOfMemoryError, stAllocObject for new class failed**

**Explanation:** The Java heap is full.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Use a larger maximum heap to start the JVM; for example, by using the Java -Mx option. In a shared classes environment, the storage area that is full depends on the type of class that is being allocated. See the sharable JVM documentation for more information about how to increase the memory that is available for class storage.

**JVMCL024 OutOfMemoryError, add package to shared NameSpace failed**

**Explanation:** Both the Java system heap and alloced memory are needed to add a package to the shared name space and one of this cannot provide enough storage to do so.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Use a larger maximum heap to start the JVM; for example, by using the Java -Mx option. In a shared classes environment, the storage area that is full depends on the type of class that is being allocated. See the sharable JVM documentation for more information about how to increase the memory that is available for class storage.

**JVMCL027 OutOfMemoryError, allocating an array of objects**

**Explanation:** An unrecoverable internal processing error has occurred in the JVM.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Use a larger maximum heap to start the JVM; for example, by using the Java -Mx option. In a shared classes environment, the storage area that is full depends on the type of class that is being allocated. See the sharable JVM documentation for more information about how to increase the memory that is available for class storage.

**JVMCL028 OutOfMemoryError, name of inner array class is NULL**

**Explanation:** An unrecoverable internal processing error has occurred in the JVM.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Contact your IBM service representative.

**JVMCL029 OutOfMemoryError, inner class name is NULL (multi-D array)**

**Explanation:** An unrecoverable internal processing error has occurred in the JVM.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Contact your IBM service representative.

**JVMCL030 OutOfMemoryError, add into loader cache failed**

**Explanation:** An unrecoverable internal processing error has occurred in the JVM.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Use a larger maximum heap to start the JVM; for example, by using the Java -Mx option. In a shared classes environment, the storage area that is full depends on the type of class that is being allocated. See the sharable JVM documentation for more information about how to increase the memory that is available for class storage.

**JVMCL031 Maximum number of shared classes exceeded, use -Xscmax command eline option to increase the limit.**

**Explanation:** This message is issued if an application attempts to load more than the specified number (or the default number, if the option was not specified) of shareable classes. These include shareable application classes, middleware classes, and system classes. A JVM set normally loads at least 1500 system classes during initialization.

**System action:** The JVM set is terminated.

**User response:** Review the number of shared classes that are being loaded by the application, and, if necessary, set the -Xscmax <n> command line option on the Master JVM to increase the limit.

**JVMCL032 OutOfMemoryError, clAddUTF8String failed**

**Explanation:** The operation to add a string to the JVM internal cache failed.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment. In a shared classes environment, see the shared classes *User Guide* for options that control shared memory, because the storage is managed by the JVM in this case.

**JVMCL033 OutOfMemoryError, creation of loader shadow failed, or promoteLoaderCaches failed**

**Explanation:** The system calloc function that was used to get storage from the system has returned NULL.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment. In a shared classes environment, see the shared classes *User Guide* for options that control shared memory, because the storage is managed by the JVM in this case.

**JVMCL034 OutOfMemoryError, sysMalloc for bigger buffer failed**

**Explanation:** The system malloc function has returned NULL.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

**JVMCL035 OutOfMemoryError, allocation failed**

**Explanation:** The loading of a Java class has failed because not enough storage is available.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Use a larger maximum heap to start the JVM; for example, by using the Java -Mx option. In a shared classes environment, the storage area that is full depends on the type of class that is being allocated. See the sharable JVM documentation for more information about how to increase the memory that is available for class storage.

**JVMCL036 OutOfMemoryError, stAllocObject failed in WRAP**

**Explanation:** The loading of a Java class has failed because not enough storage is available.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Use a larger maximum heap to start the JVM; for example, by using the Java -Mx option. In a shared classes environment, the storage area that is full depends on the type of class that is being allocated. See the sharable JVM documentation for more information about how to increase the memory that is available for class storage.

**JVMCL037 OutOfMemoryError, stAllocObject failed in WRAP2**

**Explanation:** The loading of a Java class has failed because not enough storage is available.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Use a larger maximum heap to start the JVM; for example, by using the Java -Mx option. In a shared classes environment, the storage area that is full depends on the type of class that is being allocated. See the sharable JVM documentation for more information about how to increase the memory that is available for class storage.

**JVMCL038 OutOfMemoryError, unable to allocate a loader cache entry**

**Explanation:** The JVM has been unable to allocate storage during Java class loading.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment. In a shared classes environment, see the shared classes *User Guide* for options that control shared memory, because the storage is managed by the JVM in this case.

**JVMCL039 OutOfMemoryError, failure allocating constraint spill area**

**Explanation:** The JVM has been unable to allocate storage during Java class loading. The system malloc function has returned NULL.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

**JVMCL042 OutOfMemoryError, unable to allocate NameSpace storage**

**Explanation:** The JVM has run out of memory during class loading.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment. In a shared classes environment, see the shared classes *User Guide* for options that control shared memory, because the storage is managed by the JVM in this case.

**JVMCL044 OutOfMemoryError, unable to add name space cache entry**

**Explanation:** The JVM has run out of memory during class loading.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment. In a shared classes environment, see the shared classes *User Guide* for options that control shared memory, because the storage is managed by the JVM in this case.

**JVMCL045 OutOfMemoryError, stInternString failed**

**Explanation:** The JVM has run out of memory during class loading.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment. In a shared classes environment, see the shared classes *User Guide* for options that control shared memory, because the storage is managed by the JVM in this case.

**JVMCL046 OutOfMemoryError, stInternString failed**

**Explanation:** The JVM has run out of memory during class loading.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment. In a shared classes environment, see the shared classes *User Guide* for options that control shared memory, because the storage is managed by the JVM in this case.

**JVMCL048 Illegal re-definition of class**

**Explanation:** Two concurrent attempts to load a class have caused different class objects to be resolved. The error has occurred because a race condition exists between two threads that have loaded the same class, and because an unstable underlying source of class definitions, such as .class files, has been changed.

**System action:** A Java linkageError exception is thrown.

**User response:** You can either synchronize or control the multithreaded nature of the class loading, or prevent loading from occurring while the underlying class sources are being changed. Because class loading in one class loader is synchronized, this error is more likely to occur if multiple class-loaders in a delegation chain become the defining loader for the class over time because of changes in the underlying bytecode source location.

**JVMCL049 Illegal re-definition of class**

**Explanation:** Two concurrent attempts to load a class have caused different class objects to be resolved. The error has occurred because a race condition exists between two threads that have loaded the same class, and because an unstable underlying source of class definitions, such as .class files, has been changed.

**System action:** A Java linkageError exception is thrown.

**User response:** You can either synchronize or control the multithreaded nature of the class loading, or prevent loading from occurring while the underlying class sources are being changed. Because class loading in one class loader is synchronized, this error is more likely to occur if multiple class-loaders in a delegation chain become the defining loader for the class over time because of changes in the underlying bytecode source location.

**JVMCL050 OutOfMemoryError, stInternString failed**

**Explanation:** The JVM has run out of memory during class loading.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment. In a shared classes environment, see the shared classes *User Guide* for options that control shared memory, because the storage is managed by the JVM in this case.

**JVMCL051 OutOfMemoryError, stInternString failed**

**Explanation:** The JVM has run out of memory during class loading.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment. In a shared classes environment, see the shared classes *User Guide* for options that control shared memory, because the storage is managed by the JVM in this case.

**JVMCL052 Cannot allocate memory in initializeHeap for heap segment**

**Explanation:** The malloc function that was used to initialize a Heap segment has returned NULL.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment. In a shared classes environment, see the shared classes *User Guide* for options that control shared memory, because the storage is managed by the JVM in this case.

**JVMCL053 Cannot allocate memory in allocHeap for heap segment**

**Explanation:** The malloc function that was used to allocate a Heap segment has returned NULL.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment. In a shared classes environment, see the shared classes *User Guide* for options that control shared memory, because the storage is managed by the JVM in this case.

**JVMCL054 Cannot allocate memory in allocHeap for heap segment**

**Explanation:** The malloc function that was used to allocate a Heap segment has returned NULL.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment. In a shared classes environment, see the shared classes *User Guide* for options that control shared memory, because the storage is managed by the JVM in this case.

**JVMCL055 Cannot allocate memory in initializeClassCache for context class table**

**Explanation:** The calloc function that was used to allocate memory for the class cache for the context class table has returned NULL.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment. In a shared classes environment, see the shared classes *User Guide* for options that control shared memory, because the storage is managed by the JVM in this case.

**JVMCL056 Cannot allocate memory in expandClassTable for Class cache**

**Explanation:** The calloc function that was used to allocate memory for the newly expanded class cache table has returned NULL.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment. In a shared classes environment, see the shared classes *User Guide* for options that control shared memory, because the storage is managed by the JVM in this case.

**JVMCL057 Cannot allocate memory in initializeCPTable for context shadow table**

**Explanation:** The calloc function that was used to allocate memory for the backing shadow table has returned NULL.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment. In a shared classes environment, see the shared classes User Guide for options that control shared memory, because the storage is managed by the JVM in this case.

**JVMCL200 Classloader system property**

**Explanation:** The calloc function that was used to allocate memory for the backing shadow table has returned NULL.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if

that is possible in your environment. In a shared classes environment, see the shared classes *User Guide* for options that control shared memory, because the storage is managed by the JVM in this case.

## JVM error messages for JVMDC

**JVMDC001 OutOfMemoryError, stAllocArray for cString2JavaString failed**

**Explanation:** The Java heap is full. An attempt to allocate an array of characters from the Java heap failed.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Use a larger maximum heap to start the JVM; for example, by using the java -Mx option.

**JVMDC002 OutOfMemoryError, makeByteString failed**

**Explanation:** The Java heap is full. An attempt to allocate an array of bytes from the Java heap failed.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Use a larger maximum heap to start the JVM; for example, by using the java -Mx option.

**JVMDC003 OutOfMemoryError, stAllocArray for utf2JavaString failed**

**Explanation:** The Java heap is full. An attempt to allocate an array of characters from the Java heap failed.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Use a larger maximum heap to start the JVM; for example, by using the Java -Mx option.

**JVMDC004 OutOfMemoryError, stAllocObject for utf2JavaString failed**

**Explanation:** The Java heap is full. An attempt to allocate a java/lang/String object failed.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Use a larger maximum heap to start the JVM; for example, by using the Java -Mx option.

**JVMDC005 OutOfMemoryError, stAllocArray for utfClassName2JavaString failed**

**Explanation:** The Java heap is full. An attempt to allocate an array of characters from the Java heap failed.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Use a larger maximum heap to start the JVM; for example, by using the Java -Mx option.

**JVMDC006 OutOfMemoryError, stInternString failed**

**Explanation:** During the conversion of a utf string to a Java string, the function that resolves a Java string to a single and unique equivalent inside the JVM failed and returned a NULL value because of a lack of java heap memory.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Use a larger maximum heap to start the JVM; for example, by using the Java -Mx option.

**JVMDC007 OutOfMemoryError, stAllocObject for utfClassName2JavaString failed**

**Explanation:** The Java heap is full. An attempt to allocate a java/lang/String object failed.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Use a larger maximum heap to start the JVM; for example, by using the Java -Mx option.

**JVMDC008 OutOfMemoryError, sysMalloc failed**

**Explanation:** The malloc function that was used to create a new buffer has returned NULL.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

**JVMDC009 OutOfMemoryError, sysMalloc failed**

**Explanation:** The malloc function that was used to create a buffer used while converting a platform string to a utf8 class name has returned NULL.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** The C-runtime heap of the process (not the Java object heap) is full. Increase the heap if that is possible in your environment.

## JVM error messages for JVMDBG

**JVMDBG001 malloc failed to allocate <allocation request size> bytes, time <date and time>**

**Explanation:** The system malloc function has returned NULL

**System action:** None for this message. The JVM might issue a further message.

**User response:** Increase the available native storage.

**JVMDBG002 strndup failed to allocate <allocation request size> bytes, time <date and time>**

**Explanation:** The system strndup function has returned NULL

**System action:** None for this message. The JVM might issue a further message.

**User response:** Increase the available native storage.

**JVMDBG003 strdup failed, time <date and time>**

**Explanation:** The system strdup function has returned NULL

**System action:** None for this message. The JVM might issue a further message.

**User response:** Increase the available native storage.

**JVMDBG004 calloc failed to allocate an array of <number of array elements> elements at <each array element size> bytes each, time <date and time>**

**Explanation:** The system calloc function has returned NULL

**System action:** None for this message. The JVM might issue a further message.

**User response:** Increase the available native storage.

**JVMDBG005 realloc failed to allocate <allocation request size> bytes, time <date and time>**

**Explanation:** The system realloc function has returned NULL

**System action:** None for this message. The JVM might issue a further message.

**User response:** Increase the available native storage.

# JVM error messages for JVMDG

**JVMDG009 RC %d from sysMonitorEnter in getTraceLock**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**JVMDG010 RC %d from sysMonitorExit in freeTraceLock**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**JVMDG011 RC %d from sysMonitorEnter in postWriteThread**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**JVMDG012 RC %d from sysMonitorNotify in postWriteThread**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**JVMDG013 RC %d from sysMonitorExit in postWriteThread**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**JVMDG015 Malloc failure in addTraceCmd**

**Explanation:** During processing of the user-supplied trace options, a call to malloc was made to obtain a block of memory. This call failed.

**System action:** The JVM is terminated.

**User response:** Try running the JVM with a larger maximum heap size (by using the -Xmx option). If the problem remains, contact your IBM service representative.

**JVMDG032 Unable to open properties file %s**

**Explanation:** The JVM was unable to open the properties file that was listed in the message.

**System action:** The JVM is terminated.

**User response:** Ensure that the properties file that you have specified really exists. If the problem remains, contact your IBM service representative.

**JVMDG033 Unable to determine size of properties file %s**

**Explanation:** Having opened the trace properties file mentioned in the message, the JVM was unable to determine its size.

**System action:** The JVM is terminated.

**User response:** Ensure that the file that you specified is a valid properties file, and that it is readable. If the problem remains, contact your IBM service representative.

**JVMDG034 Cannot obtain memory to process %s**

**Explanation:** To process the trace properties file, it is read into memory. Unfortunately, the call to obtain the memory for this failed.

**System action:** The JVM is terminated.

**User response:** Try running the JVM with a larger maximum heap size (by using the -Xmx option). If the problem remains, contact your IBM service representative.

**JVMDG035 Error reading properties file %s**

**Explanation:** To process the trace properties file, it is read into memory. Unfortunately, the call to read it into memory has failed.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

**JVMDG037 Unrecognized line in %s: "%s"**

**Explanation:** While reading the trace properties file, a line has been found that contains a keyword that is not recognized. The properties file name and the offending line are included in the text of the message.

**System action:** The JVM is terminated.

**User response:** Correct the line in error and try again.

**JVMDG046 RC %d from sysMonitorEnter in trace write thread**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**JVMDG047 RC %d from sysMonitorWait in trace write thread**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**JVMDG048 RC %d from sysMonitorExit in trace write thread**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**JVMDG060 Error starting trace write thread**

**Explanation:** The trace write thread is responsible for writing trace data to disk. It could not be started.

**System action:** The trace data is not written to disk.

**User response:** Ensure that you are not running into a system thread limit. If the problem remains, contact your IBM service representative.

**JVMDG070 Syntax error encountered at offset %d in:%s**

**Explanation:** The way in which you have specified which components are to have high use tracing enabled has caused a problem.

**System action:** The JVM is terminated.

**User response:** Correct the -Dibm.dg.trc.highuse=... parameter and try again.

**JVMDG078 RC %d from sysMonitorEnter in dgTraceLock**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**JVMDG079 RC %d from sysMonitorExit in dgTraceUnlock**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**JVMDG080 Cannot find class %s**

**Explanation:** The JVM was attempting to initialize the (named) Trace class, but it cannot find it.

**System action:** The JVM is terminated.

**User response:** Handle this problem in the same way that you would any other class-not-found condition.

**JVMDG081 Exception %s occurred during trace initialization**

**Explanation:** The JVM was attempting to initialize the Trace class, but its initializeTrace() method has thrown the specified exception.

**System action:** The JVM is terminated.

**User response:** Depends on the exception that was thrown.

**JVMDG082 Out of memory while processing properties file**

**Explanation:** The JVM was trying to allocate a block of memory to hold the traceFileSpec, but the malloc failed.

**System action:** The JVM is terminated.

**User response:** Try running the JVM with a larger maximum heap size (by using the -Xmx option). If the problem remains, contact your IBM service representative.

**JVMDG100 Cannot allocate memory for line terminator char(1)**

**Explanation:** The JVM was trying to malloc memory to hold a system specific line terminator character, but the malloc failed.

**System action:** The JVM terminates.

**User response:** Try running the JVM with a larger maximum heap size (by using the -Xmx option). If the problem remains, contact your IBM service representative.

**JVMDG101 Cannot allocate memory for line terminator char(2).**

**Explanation:** The JVM was trying to malloc memory to hold a system specific line terminator character, but the malloc failed.

**System action:** The JVM terminates.

**User response:** Try running the JVM with a larger maximum heap size (by using the -Xmx option). If the problem remains, contact your IBM service representative.

**JVMDG102 Cannot allocate memory for filename.**

**Explanation:** The JVM was trying to get a piece of memory to store the event output file name, but the malloc failed.

**System action:** The JVM terminates.

**User response:** Try running the JVM with a larger maximum heap size (by using the -Xmx option). If the problem remains, contact your IBM service representative.

**JVMDG103 Cannot allocate memory in dgEventQueueAdd**

**Explanation:** The JVM was trying to get a piece of memory to store an event queue item, but the malloc failed.

**System action:** The JVM terminates.

**User response:** Try running the JVM with a larger maximum heap size (by using the -Xmx option). If the problem remains, contact your IBM service representative.

**JVMDG104 Cannot allocate memory for printing a stack trace**

**Explanation:** The JVM was trying to get a piece of memory to store a Java stack trace, but the malloc failed.

**System action:** The JVM terminates.

**User response:** Try running the JVM with a larger maximum heap size (by using the -Xmx option). If the problem remains, contact your IBM service representative.

**JVMDG105 Cannot allocate memory for new event list item**

**Explanation:** The JVM was trying to get a piece of memory to store an event list item, but the malloc failed.

**System action:** The JVM terminates.

**User response:** Try running the JVM with a larger

maximum heap size (by using the -Xmx option). If the problem remains, contact your IBM service representative.

**JVMDG106 Cannot allocate memory for adding an event class**

**Explanation:** The JVM was trying to get a piece of memory to store an event class, but the malloc failed.

**System action:** The JVM terminates.

**User response:** Try running the JVM with a larger maximum heap size (by using the -Xmx option). If the problem remains, contact your IBM service representative.

**JVMDG107 Cannot allocate memory for printing a stack trace**

**Explanation:** The JVM was trying to get a piece of memory to store a Java stack trace, but the malloc failed.

**System action:** The JVM terminates.

**User response:** Try running the JVM with a larger maximum heap size (by using the -Xmx option). If the problem remains, contact your IBM service representative.

**JVMDG109 Cannot allocate memory for data in initDgData**

**Explanation:** On startup, the DG subcomponent has attempted to malloc three blocks of memory for the traceLock, traceTerminated, and writeEvent monitors. One or more of these mallocs has failed.

**System action:** The JVM terminates.

**User response:** Try running the JVM with a larger maximum heap size (by using the -Xmx option). If the problem remains, contact your IBM service representative.

**JVMDG124 Cannot allocate memory in dgRegisterDumpRoutine — Out of memory in rasDumpRegister**

**Explanation:** Each subcomponent registers its dump routine at startup so that it can be called back in the event of a Javadump. However, the malloc that was to add this routine to the list has failed.

**System action:** The JVM terminates.

**User response:** Try running the JVM with a larger maximum heap size (using the -Xmx option). If the problem remains, contact your IBM service representative.

**JVMDG125 Null method trace specification**

**Explanation:** The property -Dibm.dg.trc.methods= requires a following value that indicates which methods to trace. This value was omitted.

**System action:** The JVM terminates.

**User response:** Correct the parameter (for example, -Dibm.dg.trc.methods=*), then retry.

**JVMDG126 Length of dgMethodFmt exceeded**

**Explanation:** The specification of the methods to trace is too long for the available buffer.

**System action:** The JVM terminates.

**User response:** Use fewer characters to specify the methods that you want to trace, then retry.

**JVMDG127 Misplaced parentheses in method trace specification**

**Explanation:** The method specification cannot begin with **(** or **)**.

**System action:** The JVM terminates.

**User response:** Correct the method specification and retry.

**JVMDG128 Out of memory handling methods**

**Explanation:** Method trace requires a table of methods, but the malloc that was to create it failed.

**System action:** The JVM terminates.

**User response:** Try running the JVM with a larger maximum heap size (by using the -Xmx option). If the problem remains, contact your IBM service representative.

**JVMDG129 At least one method is required**

**Explanation:** The property -Dibm.dg.trc.methods requires a following value that indicates which methods to trace. This value was omitted.

**System action:** The JVM terminates.

**User response:** Correct the parameter (for example, -Dibm.dg.trc.methods=*) and retry.

**JVMDG130 Invalid wildcard in method trace**

**Explanation:** A wildcard has been detected at an illegal position in the methods property (″-Dibm.dg.trc.methods″).

**System action:** The JVM terminates.

**User response:** Correct the specification and retry.

**JVMDG135 Error %d from JVMPI EnableEvent**

**Explanation:** Method trace uses the JVMPI method entry and exit events. One or both of these could not be enabled.

**System action:** The JVM terminates.

**User response:** Method trace and JVMPI cannot be used at the same time. If you were attempting to do this, this error message was to be expected. If not, contact your IBM service representative.

**JVMDG136 Invalid method trace match flag %d**

**Explanation:** This message should never be issued.

**System action:** None.

**User response:** Contact your IBM service representative.

**JVMDG137 Invalid Signature type = %d SIGNATURE_VOID = %c**

**Explanation:** This message should never be issued.

**System action:** None.

**User response:** Contact your IBM service representative.

**JVMDG138 Invalid Signature type = %c**

**Explanation:** This message should never be issued.

**System action:** None.

**User response:** Contact your IBM service representative.

**JVMDG139 Cannot obtain memory for dgMethodfmt**

**Explanation:** Method trace required a block of memory, but the malloc for it failed.

**System action:** The JVM terminates.

**User response:** Try running the JVM with a larger maximum heap size (by using the -Xmx option). If the problem remains, contact your IBM service representative.

**JVMDG140 Invalid applid specified**

**Explanation:** The property -Dibm.dg.trc.applids must be followed by a list of applids, all of which must be of length greater than zero; for example, -Dibm.dg.trc.applids=applid1,,applid2 is not valid.

**System action:** The JVM terminates.

**User response:** Correct the list of applids and retry.

**JVMDG141 Out of memory handling applids**

**Explanation:** Processing the list of applids includes mallocing a block of storage, but this malloc failed.

**System action:** The JVM terminates.

**User response:** Try running the JVM with a larger maximum heap size (using the -Xmx option). If the problem remains, contact your IBM service representative.

**JVMDG142 At least one applid is required**

**Explanation:** The property -Dibm.dg.trc.applids must be followed by at least one applids.

**System action:** The JVM terminates.

**User response:** Retry, providing at least one applid.

**JVMDG147 Illegal subcomponent id in dump routine registration**

**Explanation:** An unexpected error has occurred during dump routine registration.

**System action:** This dump routine is not registered and will not be called in the event of a Javadump.

**User response:** Contact your IBM service representative.

**JVMDG148 Malloc failure in dg_main**

**Explanation:** A buffer was being malloced for use during a Javadump. The malloc failed.

**System action:** The JVM terminates.

**User response:** Try running the JVM with a larger maximum heap size (using the -Xmx option). If the problem remains, contact your IBM service representative.

**JVMDG157 Cannot obtain memory for Java stack trace**

**Explanation:** The JVM was invoked with the callstack option (-Dibm.dg.trc.callstack). On entry into a method, the saved stack of references to Java methods must be expanded, but the malloc to do this has failed.

**System action:** The JVM terminates.

**User response:** Try running the JVM with a larger maximum heap size (by using the -Xmx option). If the problem remains, contact your IBM service representative.

**JVMDG158 Error in Java call stack trace**

**Explanation:** The JVM was invoked with the callstack option (-Dibm.dg.trc.callstack). On exit from a method, a sanity check of what had been stored away has failed.

**System action:** The JVM terminates.

**User response:** Contact your IBM service representative.

**JVMDG159 Malloc failure in dg_main**

**Explanation:** The malloc of a block of memory that was required during DG subcomponent initialization has failed.

**System action:** The JVM terminates.

**User response:** Try running the JVM with a larger maximum heap size (by using the -Xmx option). If the problem remains, contact your IBM service representative.

**JVMDG160 Cannot obtain memory for utcAppFormat**

**Explanation:** In utcAppFormat, a malloc request was unable to be satisfied.

**System action:** The JVM terminates.

**User response:** This message indicates that native memory was exhausted. Increase the available memory or remove any extraneous memory usage.

**JVMDG161 Tracepoint %6.6X truncated**

**Explanation:** During the expansion of an application trace tracepoint, the string buffer length was exceeded

**System action:** The substitution of variables into the tracepoint stops at this point and the part-formatted string is returned.

**User response:** Substitute less data into the application trace point. Consider splitting it into multiple trace points or validating data before invoking the tracepoint

**JVMDG162 Could not allocate buffer**

**Explanation:** While checking whether backtrace was enabled, a malloc failure occurred.

**System action:** The JVM terminates.

**User response:** This message indicates that native memory was exhausted. Increase the available memory or remove any extraneous memory usage.

**JVMDG163 The Backtrace trace option is not supported on this platform**

**Explanation:** ″backtrace″ was specified in one of the trace options. Not all platforms support this option. This platform does not.

**System action:** The JVM terminates.

**User response:** Remove the ″backtrace″ keyword from all command-line trace options and any trace properties file in use. Then reissue the command.

**JVMDG200 Diagnostics system property %s%s%s**

**Explanation:** System properties that are associated with trace are echoed to stderr on startup so that you can be sure that they have been set.

**System action:** This message is displayed for each trace property set.

**User response:** This message is for information. It is not an error.

**JVMDG201 Keyword abbreviation too short**

**Explanation:** Trace properties can be abbreviated to three characters but no fewer.

**System action:** The JVM terminates.

**User response:** Correct your specification of trace properties in the trace properties file to use no abbreviations that are shorter than three characters. Then retry.

**JVMDG202 Invalid short form keyword**

**Explanation:** One of the supplied trace properties was not a recognized property name.

**System action:** The JVM terminates.

**User response:** Correct the property definitions and retry.

**JVMDG203 Exception occurred while running user thread**

**Explanation:** A user thread that was started by JVMRI caused an exception that has been caught.

**System action:** The user thread is terminated but the JVM continues.

**User response:** Examine the code that your user thread was running, to determine the location of the problem.

**JVMDG205 Out of memory in rasCreateThread**

**Explanation:** An attempt was made to allocate the memory that is required to create your JVMRI user thread, but the malloc failed.

**System action:** JNI_ENOMEM is returned to the calling agent.

**User response:** Try running the JVM with a larger maximum heap size (using the -Xmx option). If the problem remains, contact your IBM service representative.

**JVMDG206 Out of memory in rasCreateThread**

**Explanation:** An attempt was made to allocate the memory that is required to store the name of your JVMRI user thread, but the malloc failed.

**System action:** JNI_ENOMEM is returned to the calling agent.

**User response:** Try running the JVM with a larger maximum heap size by using the -Xmx option). If the problem remains, contact your IBM service representative.

**JVMDG207 Cannot create thread in rasCreateThread**

**Explanation:** A JVMRI call was made to create the user thread, but for it failed for unspecified reasons.

**System action:** JNI_ERR is returned to the calling agent.

**User response:** Investigate the reasons why this thread creation might have failed; for example, system limits, security settings.

**JVMDG208 Cannot create special thread in rasCreateThread**

**Explanation:** A JVMRI call was made to create the special user thread, but for it failed for unspecified reasons.

**System action:** JNI_ERR is returned to the calling agent.

**User response:** Investigate the reasons why this thread creation might have failed; for example, system limits, security settings.

**JVMDG209 No Javacore, JVM is not initialized rasGenerateJavacore**

**Explanation:** A JVMRI call to create a Javadump was received, but because the JVM has not yet finished initializing, this is not permitted.

**System action:** JNI_ERR is returned to the calling agent.

**User response:** Modify your agent so that you do not attempt to take a Javadump until the system has finished initializing.

**JVMDG210 Exception %d received during dump routine.**

**Explanation:** A JVMRI call to run a dump routine was actioned, but the dump routine found an exception.

**System action:** The dump routine will be truncated at this point.

**User response:** If the exception is persistent and unexpected, contact your IBM service representative.

**JVMDG211 Invalid component ID rasRunDumpRoutine**

**Explanation:** A JVMRI call was made to run a specified dump routine but the dump routine that was specified does not exist.

**System action:** JNI_ERR is returned to the calling agent.

**User response:** Correct the error, specifying the ID of an existing component.

**JVMDG212 Invalid component Name specified**

**Explanation:** The JVMRI call rasGetComponentDataArea() was made to get information about a specified component data area, but the specified component does not exist.

**System action:** JNI_ERR is returned to the calling agent.

**User response:** Allowable components are: "ci", "dg", "cl", "dc", "lk", "xe", "xm", and "st". The component names can also be supplied in uppercase, but NOT in mixed case.

**JVMDG213 Cannot create thread in rasStartThreads**

**Explanation:** A JVMRI call was made to start a user thread. It was deferred until initialization was complete, but now it has been actioned and has failed for unspecified reasons.

**System action:** The thread is not started.

**User response:** Investigate the reasons why this thread creation might have failed; for example, system limits, security settings.

**JVMDG214 Cannot create special thread in rasStartThreads**

**Explanation:** A JVMRI call was made to start a special user thread. It was deferred until initialization was complete, but now it has been actioned and has failed for unspecified reasons.

**System action:** The thread is not started.

**User response:** Investigate the reasons why this thread creation might have failed; for example, system limits, security settings.

---

**JVMDG215 Dump Handler has Processed %s Signal %i.**

**Explanation:** The Dump Handler has successfully handled the signal (specified). The selected dumps have been produced.

**System action:** For information only.

**User response:** The dumps that you selected (by use of the environment variable JAVA_DUMP_OPTS) should now be available for your use in debugging your problem or for sending to your IBM service representative.

---

**JVMDG217 Dump Handler is Processing a Signal - Please Wait.**

**Explanation:** A signal has been raised and is being processed by the dump handler.

**System action:** At this point, depending upon the options that have been set, Javadump, core dump, and CEEDUMP (z/OS only) can be taken. This message is for information only and does not indicate a further failure.

**User response:** None.

---

**JVMDG218 Dump Handler Caught Internal Exception %d Processing Signal %i.**

**Explanation:** An exception (specified) was found during an attempt to process the original signal (specified).

**System action:** Diagnostic information (dumps) might have been lost.

**User response:** Contact your IBM service representative.

---

**JVMDG219 Dump Handler Caught Internal Exception %d Processing SYSDUMP for Signal %i.**

**Explanation:** An exception (specified) was found during an attempt to generate a SYSDUMP for the original signal (specified). A SYSDUMP varies in nature from platform to platform; for example, a minidump on windows, a core dump on AIX.

**System action:** The SYSDUMP might have been lost.

**User response:** Contact your IBM service representative.

---

**JVMDG220 Dump Handler Caught Internal Exception %d Processing CEEDUMP for Signal %i The JVM may now be in an Unusable State.**

**Explanation:** An exception (specified) was found during an attempt to generate a CEEDUMP for the original signal (specified). This should occur only on z/OS.

**System action:** The CEEDUMP might have been lost.

**User response:** Contact your IBM service representative.

---

**JVMDG221 Dump Handler Caught Internal Exception %d Processing JAVADUMP for Signal %i.**

**Explanation:** An exception (specified) was found during an attempt to generate a JAVADUMP for the original signal (specified).

**System action:** The Javadump might have been lost.

**User response:** Contact your IBM service representative.

---

**JVMDG222 Dump Handler Caught Internal Exception %d Processing jvmpi_dump() for Signal %i.**

**Explanation:** An exception (specified) was found during an attempt to generate a JVMPI Heap Dump for the original signal (specified).

**System action:** The JVMPI Heap Dump might have been lost.

**User response:** Contact your IBM service representative.

---

**JVMDG223 Dump Handler Caught Internal Exception %d processing HEAPDUMP for Signal %i**

**Explanation:** A JVMMI heap dump event was been generated (JVMMI_SERVICE_EVENT_HEAPDUMP). During the processing of this event by the attached JVMMI agent, the named operating system signal was received.

**System action:** Dump processing continues.

**User response:** Find the cause of the failure in your JVMMI agent.

---

**JVMDG224 Invalid character(s) encountered in hex number "%s"**

**Explanation:** When a tpid clause in the system property -Dibm.dg.trc.trigger was being processed, a hexadecimal tracepoint id (tpid) was expected, but the JVM found a nonhexadecimal character.

**System action:** The JVM fails to initialize.

**User response:** Check the contents of the trigger=tpid(...) clauses. It must be of the form tpid(hexnumber) or tpid(hexnumber1-hexnumber2)

---

**JVMDG225 Hex number too long or too short ″%s″**

**Explanation:** When a tpid clause in the system property -Dibm.dg.trc.trigger was being processed, a specified tpid was found to be of incorrect length.

**System action:** The JVM fails to initialize.

**User response:** Check the contents of the trigger=tpid(...) clauses. It must be of the form tpid(hexnumber) or tpid(hexnumber1-hexnumber2), and the hexadecimal numbers must each be between one and eight digits in length.

---

**JVMDG226 Signed number not permitted in this context ″%s″**

**Explanation:** When a clause in the system property -Dibm.dg.trc.trigger was being processed, a negative delay count was found.

**System action:** The JVM fails to initialize.

**User response:** Check the contents of the trigger=tpid(...), method(...), and group(...) clauses. If a delaycount is specified, it must be a positive number.

---

**JVMDG227 Invalid character(s) encountered in decimal number ″%s″**

**Explanation:** When a clause in the system property -Dibm.dg.trc.trigger was being processed, a non-numeric character was found.

**System action:** The JVM fails to initialize.

**User response:** Check the contents of the trigger=tpid(...), method(...), and group(...) clauses. If a delaycount is specified, it must contain only the characters 0 through 9.

---

**JVMDG228 Number too long or too short ″%s″**

**Explanation:** When a clause in the system property -Dibm.dg.trc.trigger was being processed, a bad delay count was found.

**System action:** The JVM fails to initialize.

**User response:** Check the contents of the trigger=tpid(...), method(...), and group(...) clauses. If a delaycount is specified, it must be between one and eight digits long.

---

**JVMDG229 Invalid trigger action ″%s″ selected**

**Explanation:** When a clause in the system property -Dibm.dg.trc.trigger was being processed, an unrecognized action was found.

**System action:** The JVM fails to initialize.

**User response:** Check the contents of the trigger=tpid(...), method(...), and group(...) clauses. Where an action is specified, it must be one of the following: suspend, resume, suspendthis, resumethis, javadump, coredump, heapdump, or snap.

---

**JVMDG230 Invalid tpid clause, usage: tpid(tpid | tpidrange,action[,delaycount]) \n clause is: tpid(%s)**

**Explanation:** When a tpid clause in the system property -Dibm.dg.trc.trigger was being processed, the clause was found to have too many parameters.

**System action:** The JVM fails to initialize.

**User response:** Correct the tpid clause. Its format is displayed in the message.

---

**JVMDG231 Invalid tpid range - start value cannot be higher than end value**

**Explanation:** When a tpid clause in the system property -Dibm.dg.trc.trigger was being processed, the clause was found to contain an illegal tpid range.

**System action:** The JVM fails to initialize.

**User response:** Correct the tpid clause. For a tpid range, the end of the range cannot be lower than the start.

---

**JVMDG232 Out of memory processing trigger property**

**Explanation:** During an attempt to process a trace option, a malloc failed.

**System action:** The JVM fails to initialize.

**User response:** System memory (not the Java heap) is full. Close down some other running applications to save space.

---

**JVMDG233 Error occurred while activating tpid %X (rc=%d)**

**Explanation:** During an attempt to activate the displayed tpid, an error was received.

**System action:** The JVM fails to initialize.

**User response:** Ensure that the tpid is correct and retry the operation. If the problem remains, check TraceFormat.dat to ensure that the tpid is present in the build.

**JVMDG234 Out of memory processing trigger property**

**Explanation:** During an attempt to process a trace option, a malloc failed.

**System action:** The JVM fails to initialize.

**User response:** System memory (not Java the heap) is full. Close down some other running applications to save space.

**JVMDG235 WARNING: This trigger method spec results in 100+ trigger entries.\n. For performance reasons, you may want to narrow the selected scope.**

**Explanation:** The method spec that you provided on the method clause of the trigger property is very wide. It will cause much memory to be allocated for control structures.

**System action:** Information only; this message is issued, but no action is taken.

**User response:** You can ignore this warning and continue. However, to save memory, you should refine the trigger method spec to something that is more compact.

**JVMDG236 Out of memory processing trigger property**

**Explanation:** During an attempt to process a trace option, a malloc failed.

**System action:** The JVM fails to initialize.

**User response:** System memory (not the Java heap) is full. Close down some other running applications to save space.

**JVMDG237 Failure initializing triggering on methods**

**Explanation:** Trigger trace has failed to initialize.

**System action:** The JVM fails to initialize.

**User response:** Check the parameters that you supplied. Note that method trace and trigger trace do not work alongside JVMPI.

**JVMDG238 Too many parameters on trigger property method clause \n usage: method(methodSpec[,entryAction] [,exitAction][,delay])**

**Explanation:** When a method clause in the system property -Dibm.dg.trc.trigger was being processed, the clause was found to have too many parameters.

**System action:** The JVM fails to initialize.

**User response:** Correct the method clause. Its format is displayed in the message.

**JVMDG239 Method Spec on trigger property (method clause) may not be null**

**Explanation:** When the method spec that is in a method clause in the system property -Dibm.dg.trc.trigger was being processed, the clause was found to be empty.

**System action:** The JVM fails to initialize.

**User response:** Insert the desired method spec, and retry the operation.

**JVMDG240 Method spec for trigger may not include '!', '(' or ')'**

**Explanation:** When the method spec that is in a method clause in the system property -Dibm.dg.trc.trigger was being processed, the clause was found to contain illegal characters.

**System action:** The JVM fails to initialize.

**User response:** Correct the method spec, and retry the operation.

**JVMDG241 You must specify an entry action, an exit action or both.**

**Explanation:** When the method clause in the system property -Dibm.dg.trc.trigger was being processed, the clause was found to contain neither an entry action, nor an exit action.

**System action:** The JVM fails to initialize.

**User response:** Correct the method clause, and retry the operation. You must specify at least one of the two actions.

**JVMDG242 Out of memory processing trigger property**

**Explanation:** During an attempt to process a trace option, a malloc failed.

**System action:** The JVM fails to initialize.

**User response:** System memory (not the Java heap) is full. Close down some other running applications to save space.

**JVMDG243 Trigger groups clause has the following usage: \n group(<groupname>,<action>[,<delay>])**

**Explanation:** When the group clause in the system property -Dibm.dg.trc.trigger was being processed, the clause was found to be in error.

**System action:** The JVM fails to initialize.

**User response:** Change the group clause in line with

the usage text that is in the message. Then retry the operation.

**JVMDG244 Delay counts must be integer values from -99999 to +99999: \n group(%s,%s,%s)**

**Explanation:** A delay count can be -99999 through +99999. You supplied one that was not in this range.

**System action:** The JVM fails to initialize.

**User response:** Correct the group clause in your trigger property, then retry the operation.

**JVMDG245 Undefined Group ″%s″ specified in trigger property**

**Explanation:** When the group clause in the system property -Dibm.dg.trc.trigger was being processed, the clause was found to reference a group that not exist.

**System action:** The JVM fails to initialize.

**User response:** Correct the group clause, then retry the operation.

**JVMDG246 Out of memory processing trigger property**

**Explanation:** During an attempt to process a trace option, a malloc failed.

**System action:** The JVM fails to initialize.

**User response:** System memory (not the Java heap) is full. Close down some other running applications to save space.

**JVMDG247 Error occurred while activating trigger Group ″%s″**

**Explanation:** When the group clause in the system property -Dibm.dg.trc.trigger was being processed, the specified trigger group could not be processed.

**System action:** The JVM fails to initialize.

**User response:** Ensure that the group clause is correct, then retry the operation.

**JVMDG248 Zero length clause in trigger statement**

**Explanation:** One of the clauses of the trigger property was null.

**System action:** The JVM fails to initialize.

**User response:** Check your input, then retry the operation.

**JVMDG249 Malformed clause, requires ′)′ at the end: \n ″s″**

**Explanation:** Internal error. The system should never reach this point.

**System action:** The JVM fails to initialize.

**User response:** Contact your IBM service representative.

**JVMDG250 Out of memory processing trigger property**

**Explanation:** During an attempt to process a trace option, a malloc failed.

**System action:** The JVM fails to initialize.

**User response:** System memory (not the Java heap) is full. Close down some other running applications to save space.

**JVMDG251 Internal Error**

**Explanation:** None.

**System action:** The JVM fails to initialize.

**User response:** If the problem remains, contact your IBM service representative.

**JVMDG252 Empty trigger clause ″%s″ not permitted**

**Explanation:** The quoted command-line fragment is in error. A clause of the trigger property is null.

**System action:** The JVM fails to initialize.

**User response:** Correct the trigger property, then retry the operation.

**JVMDG253 Missing closing bracket(s) in trigger property**

**Explanation:** The trigger property did not end in a closing bracket.

**System action:** The JVM fails to initialize.

**User response:** Correct the brackets on the trigger property, then retry the operation.

**JVMDG254 Out of memory processing trigger property**

**Explanation:** During an attempt to process a trace option, a malloc failed.

**System action:** The JVM fails to initialize.

**User response:** System memory (not the Java heap) is full. Close down some other running applications to save space.

**JVMDG255 Usage error: trigger=([method(args)],[tpid(args)], [group(args)],...)**

**Explanation:** A null trigger property was found.

**System action:** The JVM fails to initialize.

**User response:** Correct the trigger property in line with the usage text that is in the message, then retry the operation.

**JVMDG256 Empty clauses not allowed in trigger property**

**Explanation:** A null clause was found in a trigger property.

**System action:** The JVM fails to initialize.

**User response:** Correct the trigger property, then retry the operation. This error probably occurred because you entered something like trigger=method,,tpid (that is, you entered too many commas).

**JVMDG257 Trigger clauses can be tpid, method or group. This is invalid: \n ″%s″**

**Explanation:** Clauses in the trigger property can begin with ″method(″, ″tpid(″, or ″group(″. Anything else is rejected.

**System action:** The JVM fails to initialize.

**User response:** Correct the trigger clauses, then retry the operation.

**JVMDG258 resumecount takes a (single) integer value from -99999 to +99999**

**Explanation:** The ibm.dg.trc.resumecount property is an integer value -99999 through +99999. The value that you specified was not in this range.

**System action:** The JVM fails to initialize.

**User response:** Correct the resumecount property, then retry the operation.

**JVMDG259 suspendcount takes a (single) integer value from -99999 to +99999**

**Explanation:** The ibm.dg.trc.suspendcount property is an integer value -99999 through +99999. The value that you specified was not in this range.

**System action:** The JVM fails to initialize.

**User response:** Correct the suspendcount property, then retry the operation.

**JVMDG260 resumecount and suspendcount may not both be set**

**Explanation:** You attempted to set the resumecount and suspendcount properties at the same time. This is not allowed.

**System action:** The JVM fails to initialize.

**User response:** Decide which property you want to use, then remove the other

**JVMDG261 Trace Buffer snap requested by triggered trace action**

**Explanation:** Trigger trace has requested that the internal trace buffers be flushed to a Snap file.

**System action:** The JVM fails to initialize.

**User response:** Check whether a file with a name that starts with ″Snap″ (commonly ″Snap0001.....″) is present. This file contains the most up-to-date JVM trace information.

**JVMDG262 Internal Error**

**Explanation:** None.

**System action:** The JVM fails to initialize.

**User response:** If the problem remains, contact your IBM service representative.

**JVMDG263 Internal Error**

**Explanation:** None.

**System action:** The JVM fails to initialize.

**User response:** If the problem remains, contact your IBM service representative.

**JVMDG264 Internal Error**

**Explanation:** None.

**System action:** The JVM fails to initialize.

**User response:** If the problem remains, contact your IBM service representative.

**JVMDG265 Unknown trigger action encountered (action=%d)**

**Explanation:** When an attempt is made to perform a trigger action, the stored trigger type is an unrecognized value.

**System action:** The JVM fails to initialize.

**User response:** This error should not occur. If it does, and occurs repeatedly, contact your IBM service representative.

**JVMDG266 Cannot allocate memory in dgRegisterDumpRoutine**

**Explanation:** During dump routine initialization, a block of memory is required, but the malloc has failed.

**System action:** The JVM terminates.

**User response:** Try running the JVM with a larger maximum heap size (by using the -Xmx option). If the problem remains, contact your IBM service representative.

**JVMDG267 Cannot suspend tracing from unidentified thread**

**Explanation:** A call was made to suspend the tracing for a thread, but the thread id was null.

**System action:** The JVM fails to initialize.

**User response:** This error should not occur. If it does, and occurs repeatedly, contact your IBM service representative.

**JVMDG268 Cannot resume tracing from unidentified thread**

**Explanation:** A call was made to resume the tracing for a thread, but the thread id was null.

**System action:** The JVM fails to initialize.

**User response:** This error should not occur. If it does, and occurs repeatedly, contact your IBM service representative.

**JVMDG269 Unknown JVMRAS interface version or modification level**

**Explanation:** When the JVMRAS interface (JVMRI) was being initialized, a problem occurred.

**System action:** The JVM fails to initialize.

**User response:** If the problem remains, contact your IBM service representative.

**JVMDG270 Unknown JVMRAS interface version or modification level**

**Explanation:** When the JVMRAS interface (JVMRI) was being initialized through JNI, a problem occurred.

**System action:** The JVM fails to initialize.

**User response:** If the problem remains, contact your IBM service representative.

**JVMDG271 Unknown JVMRAS interface version or modification level**

**Explanation:** When the JVMRAS interface (JVMRI) was being initialized from the HPI, a problem occurred.

**System action:** The JVM fails to initialize.

**User response:** If the problem remains, contact your IBM service representative.

**JVMDG272 No Heapdump, JVM is not initialized rasGenerateHeapdump**

**Explanation:** A JVMRI request to take a heapdump has been refused because the JVM is not yet initialized. Taking a heapdump before there is a heap, for example, would not make sense.

**System action:** No heapdump is produced, but, in other respects, operation continues as before.

**User response:** If the request came from your agent code, do not issue a heapdump request before you are sure that the JVM is initialized. You can determine that the JVM is initialized, for example, by registering a callback on the JVMMI event, JVMMI_EVENT_INIT_DONE.

**JVMDG273 Heapdump invoked by dgDumpHandler**

**Explanation:** A heap dump is being taken as part of the signal handling process.

**System action:** A heapdump is taken.

**User response:** For information only.

**JVMDG274 Dump Handler has Processed OutOfMemory**

**Explanation:** The Dump Handler has successfully handled an out-of-memory condition. The selected dumps have been produced.

**System action:** For information only.

**User response:** Investigate why your application might have run out of memory. For example, the specified heap size might be too small or references to unrequired objects are being retained, preventing them from being garbage collected.

**JVMDG275 Heapdump invoked by rasJniHeapDump**

**Explanation:** rasJniHeapDump calls a user agent to take a heapdump.

**System action:** This message is passed to JVMMI for inclusion in the JVMMI_EVENT_HEAP_DUMP detail information, to tell you why your agent has been requested to take a heap dump.

**User response:** For information only.

**JVMDG276 Heapdump invoked by rasGenerateHeapdump**

**Explanation:** rasGenerateHeapdump calls a user agent to take a heapdump.

**System action:** This message is passed to JVMMI for

inclusion in the JVMMI_EVENT_HEAP_DUMP detail information, telling you why your agent has been requested to take a heap dump.

**User response:** For information only.

---

**JVMDG277 Match counts must be integer values from -99999 to +99999: group(%s,%s,%s,%s)**

**Explanation:** On the **-Dibm.dg.trc.trigger=group(...)** property, the matchcount parameter (the fourth one) is out of range.

**System action:** The JVM terminates.

**User response:** Retry, specifying a different value for matchcount

---

**JVMDG278 Too many parameters on trigger property threshold clause usage: threshold(thresholdType,thresholdValue [,entryAction][,exitAction][,delay] [,matchcount])**

**Explanation:** There are too many parameters on the **-Dibm.dg.trc.trigger=threshold(...)** property.

**System action:** The JVM terminates.

**User response:** Correct the command and retry.

---

**JVMDG279 Threshold Type on trigger property (threshold clause) may not be null**

**Explanation:** You must specify a threshold type on the **-Dibm.dg.trc.trigger=threshold(...)** property.

**System action:** The JVM terminates.

**User response:** Correct the command and retry.

---

**JVMDG280 Threshold Value on trigger property (threshold clause) may not be null**

**Explanation:** You must specify a threshold value on the **-Dibm.dg.trc.trigger=threshold(...)** property.

**System action:** The JVM terminates.

**User response:** Correct the command and retry.

---

**JVMDG281 ThresholdType for trigger may not include '!', '(' or ')'**

**Explanation:** On the **-Dibm.dg.trc.trigger=threshold(...)** property, the threshold type must not include the listed characters.

**System action:** The JVM terminates.

**User response:** Correct the command and retry.

---

**JVMDG282 ThresholdValue for trigger may not include '!', '(' or ')'**

**Explanation:** On the **-Dibm.dg.trc.trigger=threshold(...)** property, the threshold value must not include the listed characters.

**System action:** The JVM terminates.

**User response:** Correct the command and retry.

---

**JVMDG283 You must specify an entry action, an exit action or both**

**Explanation:** No entry or exit action was specified on the **-Dibm.dg.trc.trigger=threshold(...)** property.

**System action:** The JVM terminates.

**User response:** Correct the command, adding an entry action, an exit action, or both.

---

**JVMDG284 Out of memory processing trigger property**

**Explanation:** While trying to store information about a trigger property, a malloc failed.

**System action:** The JVM terminates.

**User response:** This message indicates that native memory was exhausted. Increase the available memory or remove any extraneous memory usage.

---

**JVMDG285 Threshold direction undefined**

**Explanation:** This message should never be issued.

**System action:** The JVM terminates.

**User response:** Contact your IBM support representative.

---

**JVMDG286 Threshold direction undefined**

**Explanation:** This message should never be issued.

**System action:** The JVM terminates.

**User response:** Contact your IBM support representative.

---

**JVMDG287 Internal error.**

**Explanation:** This message should never be issued.

**System action:** Undefined.

**User response:** Contact your IBM support representative.

---

**JVMDG288 Cannot create semaphore in initDgData**

**Explanation:** DG initialization attempted to create a semaphore, but the operation failed.

**System action:** The JVM terminates.

**User response:** Investigate whether you have reached a system limit on semaphores.

**JVMDG289 Unknown Universal Trace Client interface version or modification.**

**Explanation:** Internal error.

**System action:** The JVM terminates.

**User response:** Contact your IBM support representative.

**JVMDG290 Syntax error in early trace options environment variable**

**Explanation:**

**System action:** The JVM terminates.

**User response:** Correct the command and retry.

**JVMDG291 Syntax error in early trace options environment variable**

**Explanation:** The early trace environment variable is incorrectly specified.

**System action:** The JVM terminates.

**User response:** Correct the environment variable and retry.

**JVMDG292 Unsupported trace option in early trace options environment variable**

**Explanation:** The early trace environment variable is incorrectly specified.

**System action:** The JVM terminates.

**User response:** Correct the environment variable and retry.

**JVMDG293 Dump Handler Caught Internal Exception %d Processing HEAPDUMP for Signal %i.**

**Explanation:** During the creation of a heapdump an operating system signal occurred.

**System action:** The heapdump might be truncated.

**User response:** Be aware that the heapdump might not be useful or usable.

**JVMDG294 Error writing Java core buffer to file**

**Explanation:** Javacore processing was unable to obtain the name to use for the Javacore file.

**System action:** The Javacore is written to stderr instead of to file.

**User response:** None.

**JVMDG303 JVM Requesting Java core file**

**Explanation:** The JVM has started to generate a Javadump.

**System action:** For information only.

**User response:** None.

**JVMDG304 Java core file written to %s**

**Explanation:** The JVM has finished generating a Javadump.

**System action:** For information only.

**User response:** The Javadump is now available. Its name is specified in the message.

**JVMDG305 Java core not written, unable to allocate memory for print buffer.**

**Explanation:** Javadump processing needs a large buffer to create the Javadump. If this memory is not available, a Javadump is not produced.

**System action:** Javadump processing is skipped.

**User response:** This message is for information. It is not an error.

**JVMDG306 Exception %d received during dump routine processing, section truncated.**

**Explanation:** During the creation of the Javadump, each subcomponent is asked to provide its own dump information. In one of the components, a severe error has occurred and the production of its section of the Javadump has been terminated.

**System action:** The dump information is written to the Javadump file, but the contents of the current section ends prematurely. This message is written at that point in the Javadump and the next section continues as normal.

**User response:** Review the contents of the Javadump file and, if the truncation of the section prevents you using the Javadump for its intended purpose, contact your IBM service representative.

**JVMDG307 Exception %d received during dump routine processing, dump truncated.**

**Explanation:** During creation of the Javadump, a severe error has occurred and the production of the Javadump has been terminated.

**System action:** The dump information that has been generated so far is written to the Javadump file. Following that, this message is written into the file and the file ends prematurely.

**User response:** Contact your IBM service representative.

**JVMDG308 Error writing Java core buffer to file**

**Explanation:** Writing the Javadump to the expected file has failed for unspecified reasons.

**System action:** The Javadump file is written to stderr instead.

**User response:** If this problem remains, contact your IBM service representative.

**JVMDG310 Javacore cannot be taken by a system thread because of possible deadlocks**

**Explanation:** An internal system thread has attempted to generate a Javacore (Javadump) file, but it cannot. Before the JVM can generate a Javadump, it needs to quiesce all running threads to collect coherent data. It does this by obtaining system monitors and suspending all threads except the current one and some special threads such as Garbage Collector helper threads. The Garbage Collector does a similar action when a garbage collection is done. A problem occurs when these two events coincide. Typically this can happen if a Garbage Collector helper thread finds a problem while a garbage collection is in progress. If one of these threads attempts to generate a Javacore during a garbage collection, a deadlock almost certainly occurs. This deadlock is a design restriction.

**System action:** The Javadump file is not produced and processing continues. If the reason for the failed Javadump is a fatal error, the JVM terminates.

**User response:** If this problem remains, see Chapter 27, "JVM dump initiation," on page 251.

**JVMDG311 Set JAVA_DUMP_OPTS to request a SYSDUMP if diagnostic information is required**

**Explanation:** This is an informational message that always accompanies JVMDG310.

**System action:** None.

**User response:** See Chapter 27, "JVM dump initiation," on page 251.

**JVMDG312 Dump handler forcing garbage collection for Heapdump**

**Explanation:** This is an informational message that shows that a garbage collection cycle is being performed before a Heapdump is generated.

**System action:** A garbage collection cycle is performed.

**User response:** None.

**JVMDG313 Heapdump cannot be taken by a system thread because of possible deadlocks**

**Explanation:** An internal system thread has attempted to generate a Heapdump file, but it cannot. Before the JVM can generate a Heapdump, it needs to quiesce all running threads to collect coherent data. It does this by obtaining system monitors and suspending all threads except the current one and some special threads such as Garbage Collector helper threads. The Garbage Collector does a similar action when a garbage collection is done. A problem occurs when these two events coincide. Typically this can happen if a Garbage Collector helper thread finds a problem while a garbage collection is in progress. If one of these threads attempts to generate a Heapdump during a garbage collection, a deadlock almost certainly occurs. This deadlock is a design restriction.

**System action:** The Javadump file is not produced and processing continues. If the reason for the failed Javadump is a fatal error, the JVM terminates.

**User response:** If this problem remains, see Chapter 27, "JVM dump initiation," on page 251.

**JVMDG314 Set JAVA_DUMP_OPTS to request a SYSDUMP if diagnostic information is required**

**Explanation:** This is an informational message that always accompanies JVMDG310.

**System action:** None.

**User response:** See Chapter 27, "JVM dump initiation," on page 251.

**JVMDG315 JVM Requesting Heapdump file**

**Explanation:** The JVM has started to generate a Heapdump.

**System action:** For information only.

**User response:** None.

**JVMDG316 Unable to write Heapdump - unable to create file %s**

**Explanation:** The JVM cannot open the file.

**System action:** The Heapdump is not generated, and processing continues.

**User response:** Check whether the JVM has the correct permission to write to the output file.

**JVMDG317 Error % writing Heapdump to file**

**Explanation:** The write operation failed while it was writing the Heapdump.

**System action:** The Heapdump write operation is stopped, and processing continues.

**User response:** Determine why the Heapdump write operation failed.

**JVMDG318 Heapdump file written to %s**

**Explanation:** The JVM has finished generating a Heapdump.

**System action:** For information only.

**User response:** The Heapdump is now available. Its name is specified in the message.

# JVM error messages for JVMHP

**JVMHP002 JVM requesting Transaction Dump**

**Explanation:** During handling of a fatal signal, the JVM requests that the system produces a call BCP service IEATDUMP to produce a transaction dump.

**System action:** The JVM calls BCP service IEATDUMP to produce a transaction dump.

**User response:** See the text of system message IEA820I on the system log to determine the name of the dynamically allocated dataset. The service allocates the dataset into the normal MVS filesystem, not into the z/OS UNIX HFS. If you do not want transaction dumps to be created, set code environment variable **IBM_JAVA_ZOS_TDUMP=NO**.

**JVMHP004 Transaction dump service IEATDUMP failed with rc=0x%x (%d), reason code=0x%x (%d)**

**Explanation:** The IEATDUMP service that is called by the JVM has failed. If the rc is 4, a partial dump has been taken. If the rc is higher, the dump service completely failed.

**System action:** The JVM takes other actions that are suitable for this signal.

**User response:** Console messages that have prefix IEA have been written to the system log with more information about the failure. Look up the return code and reason code in *z/OS MVS Auth Assm Services Reference ENF-IXG*, SA22-7610.

**JVMHP006 JVM requesting CEEDUMP**

**Explanation:** The JVM calls the LE Service CEE3DMP to request a formatted application level dump.

**System action:** The JVM takes other actions that are suitable for this signal.

**User response:** None.

**JVMHP008 CEE3DMP failed with msgno 0x%x (%d)**

**Explanation:** The LE service CE3DMP failed. The message prints the hex and decimal value of the `fc.tok_msgno` returned.

**User response:** No CEEDUMP was created. Look up the meaning of the msgno in the appropriate LE documentation.

**JVMHP009 Complete CEEDUMP was written to process ID %i**

**Explanation:** A complete CEEDUMP was written into the z/OS UNIX HFS. The message indicates the complete writing of the CEEDUMP.

**User response:** None

**JVMHP010 No TDUMP requested, request threshold of %d reached**

**Explanation:** The user-specified or default limit for the number of transaction dumps has been reached.

**System action:** The JVM takes no further transaction dumps.

**User response:** If a Transaction Dump is required for diagnosis of a later program check, increase the limit by using the environment variable **IBM_JAVA_ZOS_TDUMP_COUNT**.

**JVMHP012 System Transaction Dump written to %s**

**Explanation:** The JVM called the IEATDUMP macro to write a dump a dataset.

**System action:** The JVM can gather other documentation in the form of CEEDUMPs, core dumps, JAVADUMPs, or all of these.

**User response:** Examine dumps to determine cause of failure.

**JVMHP013 JVM requesting USERABEND**

**Explanation:** The JVM call the LE Service CEE3ABD

for the JAVA_DUMP_OPTS USERABEND option.

**System action:** The JVM takes a CEEDUMP.

**User response:** This message verifies that a dump was taken in response to a user request. Examine the dump to find the cause of the failure.

---

**JVMHP014 JVM requesting user dump tool [ %s ]**

**Explanation:** A system dump has been requested and the dump tool that the user has specified is being run. The user specified this tool by using the environment setting JAVA_DUMP_TOOL=

**System action:** The JVM is requesting a system dump through the user-specified dump tool.

**User response:** This message verifies that a dump was taken in response to a user request. Examine the dump to find the cause of the failure.

---

**JVMHP015 JVM passing exception to Operating System**

**Explanation:** The JVM is passing an exception to Operating System.

**System action:** None

**User response:** None; this message is for information only.

---

**JVMHP016 JVM requesting System Dump**

**Explanation:** The JVM is taking a dump either as a response to a user signal, or as the result of an exception.

**System action:** The system attempts to produce a core file.

**User response:** None; this message is for information only.

---

**JVMHP017 System Dump written to <filename>**

**Explanation:** This message follows JVMHP016, and contains the filename of the core file that was produced. The message indicates that the core file was successfully produced.

**System action:** None

**User response:** None; this message is for information only.

---

**JVMHP018 Coredump() failed with errno: <errno> <file name>**

**Explanation:** This message is the alternative message to JVMHP017. It indicates that the core file was not successfully produced. The coredump system call failed and the errno returned is contained in the message along with the filename of the core file. A file might or might not be produced, depending on the error that was found.

**System action:** None

**User response:** Determine which errno was returned, and correct the error.

---

**JVMHP020 Unable to find entry point MiniDumpWriteDump**

**Explanation:** A system dump has been requested, and the JVM is trying to generate a minidump. The dll dbghelp.dll that generates the minidump has been loaded, but the function that is needed to generate the dump MiniDumpWriteDump does not exist.

**System action:** The system cannot find the function MiniDumpWriteDump in dbghelp.dll that it needs to generate the minidump.

**User response:** Check whether the dbghelp.dll that was supplied with the JVM is the first one that is found on the system path. If it is and the error still occurs, it might have been corrupted. Reload the dbghelp.dll onto the system.

---

**JVMHP021 Unable to load DbgHelp.dll for system dump**

**Explanation:** A system dump has requested, but the JVM cannot load the dll dbghelp, which is used to generate a minidump.

**System action:** The Microsoft debug dll dbghelp cannot be loaded.

**User response:** Ensure that dbghelp.dll is available on the path. It should have been installed with the JVM into the <java_home>\jre\bin directory.

---

**JVMHP022 Error creating system dump file: %s, GetlastError = %d**

**Explanation:** The dump file that is to contain the minidump information could not be created.

**System action:** The system call to create the dump file has failed.

**User response:** Ensure that the drive has enough disk space for the dump file. (For some applications, the dump file can be large.) If disk space does not seem to be the cause of problem, make a note of the error number, and contact your IBM Service representative.

---

**JVMHP023 Creating system dump file: %s**

**Explanation:** The dump file that contains the minidump information is being written to the named file.

**System action:** The dump file is about to be written.

**User response:** None; this message is for information only.

---

**JVMHP024 Dump to %s successful**

**Explanation:** The minidump information has been written to the named file successfully

**System action:** The system call to write the minidump information to the named file has completed successfully.

**User response:** None; this message is for information only.

---

**JVMHP025 Dump to %s failed, GetLastError = %d**

**Explanation:** The writing of the minidump information to the named file has failed.

**System action:** The system call to write the minidump information to the named file has failed.

**User response:** Ensure that the drive has enough disk space for the dump file. (For some applications, the dump file can be large.) Also ensure that the version of dbghelp.dll that is shipped with the JDK is the first one that is found on the path. If neither disk space nor dbghelp.dll seem to be the cause of problem, make a note of the error number, and contact your IBM Service representative.

---

**JVMHP026 Unique dump file not created**

**Explanation:** A unique dump file name could not be created.

**System action:** The system was trying to create a unique name and a path for the dump file into which the minidump information is to be written, but the length of the combined name and path exceeds the maximum filename/path length that is allowed.

**User response:** You cannot control the filename, but you can change the path/directory name. If the directory from which the application is being run has a very long path, that path is causing the problem. By setting the environment string **IBM_JAVACOREDIR**, you can set to an alternative directory the location to which the dump file will be written.

---

**JVMHP027 Path and filename too long**

**Explanation:** The path and filename that were generated for the minidump file exceed the maximum length allowed. The message JVMHP026 is displayed immediately before this message.

**System action:** The system was trying to create a unique name and a path for the dump file into which the minidump information is to be written, but the length of the combined name and path exceeds the maximum filename/path length that is allowed.

**User response:** You cannot control the filename, but you can change the path/directory name. If the directory from which the application is being run has a very long path, that path is causing the problem. By setting the environment string **IBM_JAVACOREDIR**, you can set to an alternative directory the location to which the dump file will be written.

---

**JVMHP028 Error switching to IFA processor rc: %08x**

**Explanation:** The JVM attempted unsuccessfully to switch to an IFA (Integrated Facility for Applications) processor. This is caused by an error condition indicated by the return code shown.

**System action:** The JVM disables further attempts to switch between IFA and standard processors and continues normally.

**User response:** Contact you IBM Service representative for further information.

---

**JVMHP029 Error switching from IFA processor rc: %08x**

**Explanation:** The JVM attempted unsuccessfully to switch from an IFA (Integrated Facility for Applications) processor back to a standard processor. This is caused by an error condition indicated by the return code shown.

**System action:** The JVM disables further attempts to switch between IFA and standard processors and continues normally.

**User response:** Contact you IBM Service representative for further information.

---

**JVMHP030 Unable to switch to IFA processor - libhpi.so needs extattr +a**

**Explanation:** The JVM attempted unsuccesfully to switch to an IFA (Integrated Facility for Applications) processor. This is because the JVM library file libhpi.so requires APF-authorisation.

**System action:** The JVM disables further attempts to switch between IFA and standard processors and continues normally.

**User response:** Ensure that file libhpi.so has extended attributes set using command ″extattr +a″.

---

**JVMHP031 Malloc for <integer> bytes failed**

**Explanation:** The JVM failed to allocate the native storage required to hold details about the Java Threads.

**System action:** The JVM was about to suspend all the Java threads ahead of Garbage Collection. The thread details would be passed to the C runtime call **pthread_quiesce_and_get_np()**, which would perform the thread suspension.

**User response:** Increase the size of available storage.

**JVMHP032 Unable to UNFREEZE suspended threads, rc: <integer> errno: <integer> errno2: <integer>**

**Explanation:** The JVM was unable to unfreeze all the Java threads .

**System action:** The JVM is trying to suspend all the Java threads ahead of Garbage Collection. A call to **pthread_quiesce_and_get_np()** to suspend the threads has been unsuccessful; a further call to **pthread_quiesce_and_get_np()** to unsuspend (unfreeze) the threads before another attempt to suspend has also been unsuccessful. The JVM exits at this point.

**User response:** Check that all LE maintenance has been installed by checking symptoms against the APAR database.

**JVMHP033 Unable to suspend threads, rc: <integer> errno: <integer> errno2: <integer>**

**Explanation:** The JVM was unable to suspend all the Java threads.

**System action:** The JVM is trying to suspend all the Java threads ahead of Garbage Collection. A call to **pthread_quiesce_and_get_np()** to suspend the threads has been unsuccessful. The JVM exits at this point.

**User response:** Check that all LE maintenance has been installed by checking symptoms against the APAR database.

# JVM error messages for JVMLK

**JVMLK001 Current thread not owner**

**Explanation:** A thread has attempted to exit an inflated monitor when it does not own that monitor.

**System action:** The JVM throws a java.lang.IllegalMonitorStateException.

**User response:** Contact your IBM service representative.

**JVMLK002 Current thread not owner**

**Explanation:** A thread has attempted to exit a flat monitor when it does not own that monitor.

**System action:** The JVM throws a java.lang.IllegalMonitorStateException.

**User response:** Contact your IBM service representative.

**JVMLK003 Current thread not owner**

**Explanation:** A thread has attempted to call a notify() method on an object when it does not own the flat lock on that object.

**System action:** The JVM throws a java.lang.IllegalMonitorStateException.

**User response:** Correct the Java application code. Use a synchronized block or method so that the thread owns the lock on the object before it calls the notify() method.

**JVMLK004 Current thread not owner**

**Explanation:** A thread has attempted to call a notify() method on an object when it does not own the inflated lock on that object

**System action:** The JVM throws a java.lang.IllegalMonitorStateException.

**User response:** Correct the Java application code. Use a synchronized block or method so that the thread owns the lock on the object before it calls the notify() method.

**JVMLK005 Current thread not owner**

**Explanation:** A thread has attempted to call a notifyAll() method on an object when it does not own the flat lock on that object.

**System action:** The JVM throws a java.lang.IllegalMonitorStateException.

**User response:** Correct the Java application code. Use a synchronized block or method so that the thread owns the lock on the object before it calls the notifyAll() method.

**JVMLK006 Current thread not owner**

**Explanation:** A thread has attempted to call a notifyAll() method on an object when it does not own the inflated lock on that object.

**System action:** The JVM throws a java.lang.IllegalMonitorStateException.

**User response:** Correct the Java application code. Use a synchronized block or method so that the thread owns the lock on the object before it calls the notifyAll() method.

**JVMLK007 Operation interrupted**

**Explanation:** A thread has been interrupted during a wait() method by another thread that was calling the interrupt() method on the thread class.

**System action:** The JVM throws a java.lang.InterruptedException.

**User response:** Correct the Java application code so

that it handles the InterruptedException.

---

**JVMLK008 Current thread not owner**

**Explanation:** A thread has attempted to call a wait() method on an object when it does not own the lock on that object.

**System action:** The JVM throws a java.lang.IllegalMonitorStateException.

**User response:** Correct the Java application code. Use a synchronized block or method, so that the thread owns the lock on the object before it calls the wait() method.

---

**JVMLK010 Current thread not owner**

**Explanation:** A thread has attempted to call a wait() method on an object when it does not own the flat lock on that object.

**System action:** The JVM throws a java.lang.IllegalMonitorStateException.

**User response:** Correct the Java application code. Use a synchronized block or method, so that the thread owns the lock on the object before it calls the notifyAll() method.

---

**JVMLK011 Totally out of thread IDs**

**Explanation:** The maximum number of threads that are allowed in the JVM has been exceeded.

**System action:** The JVM is terminated.

**User response:** Reduce the number of threads that have started by the Java application.

---

**JVMLK012 Expanding monitor pool by <n> monitors to <m>**

**Explanation:** The number of inflated monitors (locks) that are required by the Java application has exceeded the number that are available in the monitor pool. The size of the pool has been increased as indicated in the message. This message is issued only if verbose monitor garbage collection mode (command line option -Xverbosemongc) is specified.

**System action:** The number of monitors that are in the pool is increased.

**User response:** This message is for information only and can be ignored.

---

**JVMLK013 Expanding monitor pool by <n> monitors to <m>**

**Explanation:** The number of inflated monitors (locks) that are available in the JVM has been increased. The hash table that is used to index the monitor pool is about to be expanded as indicated in the message. This message is issued only if verbose monitor garbage collection mode (command line option -Xverbosemongc) is specified.

**System action:** The size of the monitor pool hash table is increased as indicated.

**User response:** This message is for information only and can be ignored.

---

**JVMLK014 Monitor cache GC freed <n> of <m> monitors in <t> ms (<x> total free)**

**Explanation:** During garbage collection, a scan of the monitor cache in the JVM has found the indicated number of unused inflated monitors. This message is issued only if verbose monitor garbage collection mode (command line option -Xverbosemongc) is specified.

**System action:** The indicated number of inflated monitors are freed.

**User response:** This message is for information only and can be ignored.

---

**JVMLK015 Unlocking <o> - Flat locked when Thread <x> exited**

**Explanation:** A thread has terminated while still holding the flat lock on the object indicated. This message is issued only by the PD build.

**System action:** The flat lock on the object is released and thread termination continues.

**User response:** Contact your IBM service representative.

---

**JVMLK016 Unlocking <o> - Locked when Thread <x> exited**

**Explanation:** A thread has terminated while still holding the inflated lock on the object indicated. This message is issued only by the PD build.

**System action:** The inflated lock on the object is released and thread termination continues.

**User response:** Contact your IBM service representative.

---

**JVMLK017 obj <o> mid <m> monIndex(mid) <p1> monIndexToMonitor(monIndex(mid)) <p2>**

**Explanation:** A thread has terminated while still holding the inflated lock on the object indicated. This message is issued only by the PD build.

**System action:** An internal consistency check on object monitor pointers has failed.

**User response:** Contact your IBM service representative.

**JVMLK018 OutOfMemoryError, sysMalloc returned NULL**

**Explanation:** A system memory allocation call has failed while the JVM was initializing its internal monitors.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Increase the runtime (heap) memory that is available in the process that is running the JVM.

**JVMLK019 OutOfMemoryError, sysMalloc returned NULL**

**Explanation:** A system memory allocation call has failed while the JVM was initializing its internal monitors.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Increase the runtime (heap) memory that is available in the process that is running the JVM.

**JVMLK020 Cannot allocate memory for micb table in monPoolInit**

**Explanation:** A system memory allocation call has failed while the JVM was initializing its monitor pool.

**System action:** The JVM is terminated.

**User response:** Increase the runtime (heap) memory that is available in the process that is running the JVM.

**JVMLK021 Cannot allocate memory for monitor buffer monPoolExpand**

**Explanation:** A system memory allocation call has failed while the JVM was expanding its monitor pool.

**System action:** The JVM is terminated.

**User response:** Increase the runtime (heap) memory that is available in the process that is running the JVM.

**JVMLK022 Cannot allocate memory for new buffer in monPoolExpand**

**Explanation:** A system memory allocation call has failed while the JVM was expanding its monitor pool.

**System action:** The JVM is terminated.

**User response:** Increase the runtime (heap) memory that is available in the process that is running the JVM.

**JVMLK023 Cannot allocate memory in inflMonitorInit**

**Explanation:** A system memory allocation call has failed while the JVM was initializing an inflated monitor.

**System action:** The JVM is terminated.

**User response:** Increase the runtime (heap) memory that is available in the process that is running the JVM.

**JVMLK024 Failed to obtain local monitor**

**Explanation:** Internal error.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

**JVMLK025 Failed to obtain global monitor**

**Explanation:** Internal error.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

**JVMLK026 Failed to release global monitor**

**Explanation:** Internal error.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

**JVMLK027 Failed to release local monitor**

**Explanation:** Internal error.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

**JVMLK028 Failed to obtain local monitor**

**Explanation:** Internal error.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

**JVMLK029 Failed to release local monitor**

**Explanation:** Internal error.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

**JVMLK030 Cannot allocate memory in lkGetLocalProxy()**

**Explanation:** An out-of-memory condition occurred while memory was being allocated for shared JVM locks.

**System action:** The JVM is terminated.

**User response:** Increase the runtime (heap) memory that is available in the process that is running the JVM.

# JVM error messages for JVMST

**JVMST001 Cannot allocate memory in initWorkPackets**

**Explanation:** Not enough virtual storage was available to allocate the concurrent data structures. The call to sysMalloc() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMST010 Cannot allocate memory for ACS area**

**Explanation:** Not enough virtual storage was available to allocate the ACS heap. The call to sharedMemoryAlloc() failed. This can happen during the initialization or expansion of the ACS heap.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMST011 JVMST011**

**Explanation:** Not enough virtual storage was available to allocate the mirrored card table. The call to sysMapMem() failed. This can happen only in the debug build during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMST012 Cannot allocate memory in concurrentInit()**

**Explanation:** Not enough virtual storage was available to allocate the stop_the_world_mon monitor. The call to sysMalloc() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMST013 Cannot allocate memory in initGcHelpers(2)**

**Explanation:** Not enough virtual storage was available to allocate the ack_mon monitor. The call to sysMalloc() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMST014 Cannot allocate memory in initConBKHelpers(3)**

**Explanation:** Not enough virtual storage was available to start a concurrent background thread. The call to xmCreateSystemThread() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMST015 Cannot commit memory in initConcurrentRAS**

**Explanation:** An error occurred during an attempt to commit memory for the mirrored card table. The call to sysCommitMem() failed. This only happen only in the debug build during initialization.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

**JVMST016 Cannot allocate memory for initial Java heap**

**Explanation:** Not enough virtual storage was available to allocate the Java heap. The call to sysMapMem() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative

**JVMST017 Cannot allocate memory in initializeMarkAndAllocBits(markbits1)**

**Explanation:** Not enough virtual storage was available to allocate the markbits vector. The call to sysMapMem() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMST018 Cannot allocate memory for initializeMarkAndAllocBits(allocbits1)**

**Explanation:** Not enough virtual storage was available to allocate the allocbits vector. The call to sysMapMem() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMST019 Cannot allocate memory in allocateToMiddlewareHeap**

**Explanation:** An error occurred during an attempt to commit memory for the Java heap. The call to sysCommitMem() failed. This can happen during initialization or during expansion of the heap.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

**JVMST020 Cannot allocate memory in allocateToTransientHeap**

**Explanation:** An error occurred during an attempt to commit memory for the transient heap. The call to sysCommitMem() failed. This can happen during initialization or during expansion of the transient heap.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

**JVMST021 Cannot allocate memory in initParallelMark(stackEnd**

**Explanation:** Not enough storage was available in the Java heap to allocate the stackEnd object. The call to allocMiddlewareArray() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more Java heap storage by increasing the -Xmx value. If the problem remains, contact your IBM service representative.

**JVMST022 Cannot allocate memory in initParallelMark(pseudoClass**

**Explanation:** Not enough storage was available in the Java heap to allocate the pseudoClass object. The call to allocMiddlewareObject() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more Java heap storage by increasing the -Xmx value. If the problem remains, contact your IBM service representative.

**JVMST023 Cannot allocate memory in initializeGCFacade**

**Explanation:** Not enough virtual storage was available to allocate the verbosegc buffer. The call to sysMalloc() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMST024 Cannot allocate memory in concurrentInit(base-Malloc)**

**Explanation:** Not enough virtual storage was available to allocate the concurrent data structures. The call to sysMalloc() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative

**JVMST025 Cannot allocate memory in icDoseThread**

**Explanation:** Not enough virtual storage was available to allocate a sys_thread_stack_segment. The call to sysCalloc() failed. This can happen only during garbage collection.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMST026 Cannot allocate memory in initializeMiddlewareHeap (not enough memory)**

**Explanation:** An error occurred during an attempt to allocate storage to the middleware heap. The call to allocateToMiddlewareHeap() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

**JVMST027 Cannot allocate memory for System Heap area in allocateSystemHeapMemory**

**Explanation:** Not enough virtual storage was available to allocate storage for the system heap. The call to

sharedMemoryAlloc() failed. This can happen during initialization or during expansion of the system heap.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

---

**JVMST028 Cannot commit memory in RASinitShadowHeap**

**Explanation:** An error occurred during an attempt to commit memory for the shadow heap. The call to sysCommitMem() failed. This can happen only during initialization when the trace option -Dibm.dg.trc.print=st_concurrent_shadow_heap is used.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

---

**JVMST029 Cannot allocate memory in jvmpi_scan_thread_roots**

**Explanation:** Not enough virtual storage was available to allocate a sys_thread_stack_segment. The call to sysCalloc() failed. This can happen only during garbage collection when JVMPI is running.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

---

**JVMST030 Cannot allocate memory in initializeCardTable**

**Explanation:** Not enough virtual storage was available to allocate the card table. The call to sysMapMem() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

---

**JVMST031 Cannot commit memory in initializeCardTable**

**Explanation:** An error occurred during an attempt to commit memory for the card table. The call to sysCommitMem() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

---

**JVMST032 Cannot allocate memory in initializeTransientHeap**

**Explanation:** An error occurred during an attempt to allocate storage to the transient heap. The call to allocateToTransientHeap() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

---

**JVMST033 Cannot allocate memory in initializeMarkAndAllocBits(markbits2)**

**Explanation:** An error occurred during an attempt to commit memory for the markbits vector. The call to sysCommitMem() failed. This can happen only during initialization when -Xresettable is running.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

---

**JVMST034 Cannot allocate memory in initializeMarkAndAllocBits(allocbits2)**

**Explanation:** An error occurred during an attempt to commit memory for the allocbits vector. The call to sysCommitMem() failed. This can happen only during initialization when -Xresettable is running.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

---

**JVMST035 Cannot allocate memory in initializeMiddlewareHeap (markbits)**

**Explanation:** An error occurred during an attempt to commit memory for the markbits vector. The call to sysCommitMem() failed. This can happen only during initialization when -Xresettable is not running.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

---

**JVMST036 Cannot allocate memory in initializeMiddlewareHeap (allocbits)**

**Explanation:** An error occurred during an attempt to commit memory for the allocbits vector. The call to sysCommitMem() failed. This can happen only during initialization when -Xresettable is not running.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

**JVMST039 Cannot allocate Shared Memory segment in initializeSharedMemory**

**Explanation:** An error occurred during an attempt to create shared memory. The call to xhpiSharedMemoryCreate() failed. This can happen only during initialization when -Xjvmset is running.

**System action:** A return code of JNI_ENOMEM is passed back to the JNI_CreateJavaVM call.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMST040 Cannot initialize Java heap in allocateToMiddlewareHeap**

**Explanation:** An error occurred during an attempt to commit memory for the Java heap. The call to sysCommitMem() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

**JVMST042 Cannot allocate memory in initParallelMark(base-Malloc)**

**Explanation:** Not enough virtual storage was available to allocate the parallel mark data structures. The call to sysMalloc() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMST043 Cannot allocate memory in concurrentScanThread**

**Explanation:** Not enough virtual storage was available to allocate a sys_thread_stack_segment. The call to sysCalloc() failed. This can happen only during concurrent marking.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMST044 Cannot allocate memory in concurrentInitLogCleaning**

**Explanation:** Not enough virtual storage was available to allocate the cleanedbits vector. The call to sysMapMem() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMST045 Cannot commit memory in concurrentInitLogCleaning**

**Explanation:** An error occurred during an attempt to commit memory for the cleanedbits. The call to sysCommitMem() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

**JVMST046 Cannot allocate storage for standalone jab in initializeSharedMemory**

**Explanation:** Not enough virtual storage was available to allocate the JAB. The call to sysCalloc() failed. This can happen only during initialization when -Xjvmset is not running.

**System action:** A return code of JNI_ENOMEM is be passed back to the JNI_CreateJavaVM call.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMST047 Cannot allocate memory in initParallelSweep**

**Explanation:** Not enough virtual storage was available to allocate the parallel sweep data structure PBS_ThreadStat. The call to sysMalloc() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMST048: Could not establish access to shared storage in openSharedMemory**

**Explanation:** An error occurred during an attempt to access shared memory. The call to xhpiSharedMemoryOpen() failed. This can happen only during initialization when -Xjvmset is running.

**System action:** A return code of JNI_ENOMEM is passed back to the JNI_CreateJavaVM call.

**User response:** Check that the correct token is being passed in the JavaVMOption. If the problem remains, contact your IBM service representative.

**JVMST049**

**Worker and Master JVM versions differ**
**Worker JVM version is <version> build type is <build>**
**Master JVM version is <version> build type is <build>**
**Where version is the JVM version (for example 1.3) and build is the build type (DEV, COL, or INT).**

**Explanation:** A mismatch has occurred between the Master JVM and a Worker JVM. This can happen only during initialization when -Xjvmse is running.

**System action:** A return code of JNI_ERR is passed back to the JNI_CreateJavaVM call.

**User response:** Ensure that the Master and all Worker JVMs are at the same version level, and all are of the same build type. If the problem remains, contact your IBM service representative.

---

**JVMST050 Cannot allocate memory for initial Java heap**

**Explanation:** An error occurred during an attempt to query memory availability. The call to DosQuerySysInfo() failed. This can happen only during initialization on OS/2.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

---

**JVMST051 Cannot allocate memory for initial Java heap**

**Explanation:** Not enough virtual storage was available to allocate the Java heap. The call to sysMapMem() failed. This can happen only during initialization on OS/2.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

---

**JVMST052 Cannot allocate memory for initial Java heap**

**Explanation:** Not enough virtual storage was available to allocate the Java heap. The call to sysMapMem() failed. This can happen only during initialization on OS/2 and when JAVA_HIGH_MEMORY has been specified.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

---

**JVMST055 Cannot allocate memory in initParallelSweep**

**Explanation:** Not enough virtual storage was available to allocate the parallel sweep data structure pbs_scoreboard. The call to sysMalloc() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

---

**JVMST056 Cannot allocate memory in initConBKHelpers(1)**

**Explanation:** Not enough virtual storage available to allocate the bk_activation_mon monitor. The call to sysMalloc() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

---

**JVMST057 Cannot allocate memory in initGcHelpers(1)**

**Explanation:** Not enough virtual storage was available to allocate the request_mon monitor. The call to sysMalloc() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

---

**JVMST058 Cannot allocate memory in initGcHelpers(3)**

**Explanation:** Not enough virtual storage available to start a gcHelper thread. The call to xmCreateSpecialSystemThread() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

---

**JVMST059 Cannot allocate memory in scanThread**

**Explanation:** Not enough virtual storage was available to allocate a sys_thread_stack_segment. The call to sysCalloc() failed. This can happen only during garbage collection.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMST060 Cannot allocate memory in concurrentInit() - bk_threads-sysCalloc**

**Explanation:** Not enough virtual storage was available to allocate concurrent background thread(s). The call to sysCalloc() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMST061 Cannot allocate memory in concurrentInit**

**Explanation:** Not enough virtual storage was available to allocate the concurrent tracer_mon monitor. The call to sysMalloc() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMST062 Cannot allocate memory in initializeFRBits**

**Explanation:** Not enough virtual storage was available to allocate the FRBits. The call to sysMapMem() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMST063 Cannot allocate memory in initializeFRBits**

**Explanation:** Not enough virtual storage was available to commit the FRBits in resettable code. The call to sysCommitMem() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

**JVMST064 Cannot allocate memory in initializeMiddlewareHeap**

**Explanation:** Not enough virtual storage was available to commit the FRBits in resettable code. The call to sysCommitMem() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMST065 Cannot allocate memory for break tables in initializeIncrementalCompaction**

**Explanation:** Not enough virtual storage was available to create the break tables for incremental compaction. The call to sysMalloc() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMST066 Exception (sysGetExceptionCode()) received during openSharedMemory with token(token)**

**Explanation:** Cannot access shared storage that is defined by the token that xhpiSharedMemoryOpen returns. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Check whether the token that is being passed by -Xjvmset is valid. If the problem remains, contact your IBM service representative.

**JVMST067 Invalid method_type detected in heap allocation(allocObject)**

**Explanation:** A class type that was detected during object allocation was not Middleware, Primordial, or Application.

**System action:** The JVM is terminated.

**User response:** If the problem remains, contact your IBM service representative.

**JVMST068 Invalid method_type detected in heap allocation (allocArray)**

**Explanation:** A class type that was detected during array allocation was not Middleware or Application.

**System action:** The JVM is terminated.

**User response:** If the problem remains, contact your IBM service representative.

---

**JVMST069 Invalid method_type detected in heap allocation (allocConextArray)**

**Explanation:** A class type that was detected during context array allocation was not Middleware or Application.

**System action:** The JVM is terminated.

**User response:** If the problem remains, contact your IBM service representative.

---

**JVMST070 Invalid method_type detected in heap allocation (allocConextObject)**

**Explanation:** A class type that was detected during context object allocation was not Middleware or Application.

**System action:** The JVM is terminated.

**User response:** If the problem remains, contact your IBM service representative.

---

**JVMST080 -verbose:gc flag is set**

**Explanation:** The JVM was started with switch **-verbose:gc**

**System action:** During garbage collection cycles, informational messages are written to stderr (or to a file that you specify).

**User response:** None.

---

**JVMST081 File open failed for verbose:gc output file %s**

**Explanation:** An error occurred during the opening of the file for verbose:gc output.

**System action:** The JVM directs verbose garbage collection messages to stderr instead. JVM initialization continues.

**User response:** Review stderr for verbose:gc messages. Ensure that environment variable **IBM_JVM_ST_VERBOSEGC_LOG** specifies a valid filename.

---

**JVMST082 -verbose:gc output will be written to %s**

**Explanation:** verbose:gc output will be written to the names HFS file.

**System action:** The JVM sends verbose:gc messages to the names file. JVM initialization continues.

**User response:** Review stderr for verbose:gc messages.

---

**JVMST083 Exception occurred while calculating freeList size for JVMMI**

**Explanation:** An exception occurred while the jvmmiOutOfMemoryEvent was being set up.

**System action:** The JVM is terminated.

**User response:** If the problem remains, contact your IBM service representative.

---

**JVMST084 Cannot allocate memory in stInit for segment_info**

**Explanation:** Not enough virtual storage was available to create the sys_thread_stack_segment. The call to sysCalloc() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

---

**JVMST085 Cannot suspend threads in gc0**

**Explanation:** An attempt by xmSuspendAllThreads to lock all threads before garbage collection was not successful.

**System action:** The JVM is terminated.

**User response:** If the problem remains, contact your IBM service representative.

---

**JVMST086 verifyHeap() not supported in single thread mode**

**Explanation:** ST facade function for heap verification called during GC.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

---

**JVMST088 Cannot allocate memory in "initializeSCCardTable"**

**Explanation:** Not enough virtual storage was available to allocate the shared class card table. The call to sysMapMem() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

---

**JVMST089 Cannot commit memory in ″initializeSCCardTable″**

**Explanation:** An error occurred during an attempt to commit memory for the shared class card table. The call to sysCommitMem() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

**JVMST090 Incorrect usage of -Xverbosegclog**

**Explanation:** The parameters that were passed with -Xverbosegclog are not correct.

**System action:** The JVM is terminated.

**User response:** Refer to the information about -Xverbosegclog (see Appendix G, “Command-line parameters,” on page 487). If the problem remains, contact your IBM service representative.

**JVMST092 Cannot allocate memory in initializeGCFacade**

**Explanation:** Not enough virtual storage was available to allocate the verbosegc trace buffer. The call to sysMalloc() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMST093 file open failed for verbose:gc output file**

**Explanation:** Cannot open the verbosegc log file.

**System action:** Verbosegc log output is written to the stderr log.

**User response:** Check whether the entered file name is valid and whether open is a valid operation on this file.

**JVMST094 file open failed for verbose:gc output file**

**Explanation:** Cannot open the verbosegc log file.

**System action:** Verbosegc log output is written to the stderr log.

**User response:** Check whether the entered file name is valid and whether open is a valid operation on this file.

**JVMST095 Incorrect usage of -Xverbosegclog**

**Explanation:** The parameters that were passed with -Xverbosegclog are not correct.

**System action:** The JVM is terminated.

**User response:** Refer to the information about -Xverbosegclog (see Appendix G, “Command-line parameters,” on page 487). If the problem remains, contact your IBM service representative.

**JVMST096 Out of memory in setVerbosegcRedirectionFormatScreen**

**Explanation:** Not enough virtual storage was available to allocate the verbosegc buffer. The call to sysMalloc() failed. This can happen only during initialization.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMST097 Concurrent GC is disabled**

**Explanation:** An attempt has been made to turn on concurrent verbosegc by using the dynamic switching interface when concurrent gc is not enabled.

**System action:** The JVM is terminated.

**User response:** Review the dynamic switching interface.

**JVMST099 Live Memory count inaccuracy, traced %d bytes, STW %d bytes**

**Explanation:** The counter that tracks total number of bytes consumed by live objects contains inaccurate value. This inaccurate value can affect the heuristics used in some GC algorithms.

**System action:** The JVM is terminated.

**User response:** If the problem remains, contact your IBM service representative.

**JVMST100 Unable to allocate an array object. Array element exceeds IBM JDK limit of 268435455 elements**

**Explanation:** The array could not be allocated in the requested heap because the size requested for the array exceeds the maximum size that is permitted by the IBM JDK.

**System action:** JVM terminated.

**User response:** Reduce the array size to less than 256 MB.

**JVMST101 Unable to allocate an array object. Array element exceeds IBM JDK limit of 268435455 elements**

**Explanation:** The array could not be allocated in the middleware heap because the size requested for the array exceeds the maximum size that is permitted by the IBM JDK.

**System action:** JVM terminated.

**User response:** Reduce the array size to less than 256 MB.

**JVMST102 Unable to allocate an array object. Array element exceeds IBM JDK limit of 268435455 elements**

**Explanation:** The array could not be pinned and allocated from the pinned cluster because the size requested for the array exceeds the maximum size that is permitted by the IBM JDK.

**System action:** JVM terminated.

**User response:** Reduce the array size to less than 256 MB.

**JVMST103 Unable to allocate an array object. Array element exceeds IBM JDK limit of 268435455 elements**

**Explanation:** The array could not be allocated from the middleware or transient heap because the size requested for the array exceeds the maximum size that is permitted by the IBM JDK.

**System action:** JVM terminated.

**User response:** Reduce the array size to less than 256 MB.

**JVMST104 Unable to allocate an array object. Array element exceeds IBM JDK limit of 268435455 elements**

**Explanation:** The array could not be allocated from the middleware or transient heap according to the current method context because the size requested for the array exceeds the maximum size that is permitted by the IBM JDK.

**System action:** JVM terminated.

**User response:** Reduce the array size to less than 256 MB.

**JVMST105 Unable to allocate an array object. Array element exceeds IBM JDK limit of 268435455 elements**

**Explanation:** The array could not be allocated from the same heap as the object that has been passed as one of the parameters, because the size requested for the array exceeds the maximum size that is permitted by the IBM JDK.

**System action:** JVM terminated.

**User response:** Reduce the array size to less than 256 MB.

**JVMST106 Unable to allocate an object. Object size is bigger than 1073741824 bytes**

**Explanation:** Cannot allocate a chunk of storage to use as an object, array, or one of a variety of 'special' objects such as pinned object clusters.

**System action:** JVM terminated.

**User response:** Reduce the object size to less than 1 GB.

**JVMST107 Unable to allocate an object. Object size is bigger than 1073741824 bytes**

**Explanation:** Cannot allocate a chunk of storage to use as an object, array, or one of a variety of 'special' objects such as pinned object clusters in the target heap specified.

**System action:** JVM terminated.

**User response:** Reduce the object size to less than 1 GB.

**JVMST108 Insufficient space in Java heap to satisfy allocation request**

**Explanation:** Cannot allocate a chunk of storage for use in the Java heap when concurrent is enabled.

**System action:** JVM terminated.

**User response:** Contact your IBM service representative.

**JVMST109 Insufficient space in Java heap to satisfy allocation request**

**Explanation:** Cannot allocate a chunk of storage for use in the Java heap when concurrent is disabled.

**System action:** JVM terminated.

**User response:** Contact your IBM service representative.

**JVMST110 Insufficient space in transient Java heap to satisfy allocation request**

**Explanation:** Cannot allocate a chunk of storage for use in the Java heap.

**System action:** JVM terminated.

**User response:** Contact your IBM service representative.

# JVM error messages for JVMXE

**JVMXE001 OutOfMemoryError, stAllocObject failed**

**Explanation:** The JVM cannot create a new java (class) object.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Review the memory that the application requires.

**JVMXE002 OutOfMemoryError, xeCreateStack failed**

**Explanation:** The JVM cannot create a new stack to expand an existing Java stack.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Review the Java stack size that is specified for the application. Increase the Java stack size through the -Xss parameter if required.

**JVMXE003 OutOfMemoryError, stAllocObject for executeJava failed**

**Explanation:** The JVM cannot create a new Java (class) object.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Review the memory that the application requires.

**JVMXE004 OutOfMemoryError, stAllocArray for executeJava failed**

**Explanation:** The JVM cannot create a new Java (newArray_quick) object.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Review the memory that the application requires.

**JVMXE005 OutOfMemoryError, multiArrayAlloc for executeJava failed. Unable to create the Java (multiArray) object due to Out_Of_Memory condition (inside interpreter)**

**Explanation:** The JVM cannot create a new Java (multiArray) object.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Review the memory that the application requires.

**JVMXE006 OutOfMemoryError, stAllocArray for executeJava failed**

**Explanation:** The JVM cannot create a new Java (newArray) object.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Review the memory that the application requires.

**JVMXE007 OutOfMemoryError, multiArrayAlloc failed for x86_multianewarray_quick**

**Explanation:** The JVM cannot create a new Java (newMultiArray) object.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Review the memory that the application requires.

**JVMXE008 Cannot allocate memory for temporary array in remapLocals**

**Explanation:** The JVM cannot allocate a temporary array for remapping of Locals.

**System action:** The JVM issues an ″Out of memory, aborting″ message, and is terminated.

**User response:** Review the memory that the application requires.

**JVMXE015 Cannot allocate memory for XeData in getXeDataAddress**

**Explanation:** The JVM cannot allocate memory for a Xedata area in a multi-JVM environment.

**System action:** The JVM issues an ″Out of memory, aborting″ message, and is terminated.

**User response:** Review the memory that the application requires.

**JVMXE016 Invalid JIT setting for Worker JVM**

**Explanation:** A Worker JVM has been started with a -Djava.compiler value that is different from the one that is specified for the Master JVM in a multi-JVM environment.

**System action:** The JVM issues the message and continues processing.

**User response:** Review and correct the start up parameter -Djava.compiler for the Worker JVM.

**JVMXE017 JVM will terminate at user request, Exception match**

**Explanation:** The JVM has received a signal to terminate for an exception.

**System action:** The JVM issues the message and is terminated.

**User response:** Review the exception request from the application, and correct the detected error.

**JVMXE018 StackOverflowError, expandJavaStack failed due to Insufficient Java stack size**

**Explanation:** Java Stack usage exceeded the limit or size of the Java stack.

**System action:** The JVM throws java.lang.StackOverflowError.

**User response:** Check for any Recursions, unreleased JNI references.

# JVM error messages for JVMXM

**JVMXM001 OutOfMemoryError, can't create new thread**

**Explanation:** This message is issued if the JVM cannot create a new system thread.

**System action:** The JVM throws a java.lang.OutOfMemoryError.

**User response:** Review the number of threads and memory that the application requires.

**JVMXM002 Cannot set resettable mode in a Worker JVM**

**Explanation:** This message is issued if a Worker JVM is started with the -Xresettable option.

**System action:** The JVM is terminated.

**User response:** Do not specify the -Xresettable option for a Worker JVM, because it always inherits the resettable option from the Master.

**JVMXM003 Exception %d Caught during Abort Processing**

**Explanation:** Here, %d is a JVM exception code. This message is issued during the abort of a failed JVM, if the abort processing itself finds a secondary failure.

**System action:** The JVM is aborted, but the abort processing is incomplete.

**User response:** None. The usual cause of this error is that the original JVM failure was severe enough to cause the abort processing to fail.

**JVMXM004 Error in global locking initialization**

**Explanation:** A master or worker JVM has failed to initialize because it cannot create or access the cross-process semaphores that are needed in a shareable JVM environment. Possible reasons for this message are:

- Not enough system semaphores are available. Each master JVM gets a set of 32 semaphores that its worker JVMs subsequently use.
- You are running shareable JVMs from different user IDs, and the default file permissions that are used for semaphore files are not correct.

**System action:** The JVM is terminated.

**User response:**

1. Use the **ipcs** and **ipcs -y** commands to check the use and availability of system semaphores, as follows:
   - The MNIDS value must be enough for the maximum number of semaphore IDs that are in use at one time. Each master JVM uses a single semaphore ID; other processes might use more. You can change the MNIDS value by using the IPCSEMNIDS parameter that is in the BPXPRM parmlib.
   - The MNSEMS value must be enough for the maximum number of semaphores that are requested in a semaphore set. The master JVM requests a set of 32 semaphores, so the MNSEMS value must be 32 or greater. You can change the MNSEMS value by using the IPCSEMNSEMS parameter that is in the BPXPRM parmlib. For additional information see *z/OS V1R2.0 UNIX System Services Programming: Assembler Callable Services Reference* .
2. Check whether the user's umask setting is 011. This sets default permissions of new files to rw-rw-rw-.

**JVMXM005 Unable to initialize threads**

**Explanation:** The JVM could not initialize the main thread.

**System action:** The JVM is terminated.

**User response:** Review system resources and Java heap settings. They might be too small. Review the Java installation.

**JVMXM006 Unable to initialize signal handler, thread create failed**

**Explanation:** Cannot create the signal dispatcher thread.

**System action:** The JVM is terminated.

**User response:** Review system resources and the Java installation.

**JVMXM007 Error occurred while initializing System or Runtime class**

**Explanation:** The initialization of mirrored system classes has failed. (This error is applicable to shared classes only.)

**System action:** The JVM is terminated.

**User response:** Review system resources and the Java installation. Other system messages that give more specific information might accompany this message. Running with -Xverbose might also be helpful.

**JVMXM008 Error occurred while initializing System Class**

**Explanation:** The Java System class initialization method has failed.

**System action:** The JVM is terminated.

**User response:** Review system resources and the Java installation. Other system messages that give more specific information might accompany this message.

**JVMXM009 Error occurred while initializing extra classes**

**Explanation:** Additional class initialization for shared classes mode has failed.

**System action:** The JVM is terminated.

**User response:** Review system resources and the Java installation. Other system messages that give more specific information might accompany this message.

**JVMXM010 Cannot allocate memory in eeReserveSlot()**

**Explanation:** Internal JVM request for thread data area has failed due to out of memory condition.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMXM011 Cannot allocate memory in xmPush()**

**Explanation:** An internal JVM request has failed because of an out-of-memory condition.

**System action:** The JVM is terminated.

**User response:** Allocate more virtual storage to the JVM region. If the problem remains, contact your IBM service representative.

**JVMXM012 Error occurred in diagnostics initialization(2)**

**Explanation:** The JVM RAS trace component initialization has failed.

**System action:** The JVM is terminated.

**User response:** Review system resources and the Java installation. Other system messages that give more specific information might accompany this message.

# Universal Trace Engine error messages

**UTE001 Error starting trace write thread**

**Explanation:** The trace write thread is responsible for writing trace data to disk. It could not be started.

**System action:** The trace data is not written to disk

**User response:** Ensure that you are not running into a system thread limit. If the problem remains, contact your IBM service representative.

**UTE002 Cannot open trace control file: %s**

**Explanation:** As part of initialization, the Trace Control File is loaded. This time, it cannot be opened.

**System action:** The JVM is terminated.

**User response:** If the problem remains, contact your IBM service representative.

**UTE003 Cannot obtain size of trace control file: %s**

**Explanation:** As part of initialization, when the Trace Control File is loaded, its size is calculated. This time, querying the file size has returned an error.

**System action:** The JVM is terminated.

**User response:** If the problem remains, contact your IBM service representative.

**UTE004 Trace control file %s is too large**

**Explanation:** As part of initialization, the Trace Control File is loaded into memory. However, the file is too large.

**System action:** The JVM is terminated.

**User response:** If the problem remains, contact your IBM service representative.

**UTE005 Out of memory condition processing %s**

**Explanation:** As part of initialization, the Trace Control File is loaded into memory. However, the attempt to allocate a block of memory to contain the file contents has failed because of a lack of memory.

**System action:** The JVM is terminated.

**User response:** Try running the JVM with a larger maximum heap size (by using the **-Xmx** option). If the problem remains, contact your IBM service representative.

**UTE006 Error reading %s**

**Explanation:** As part of initialization, the Trace Control File is loaded into memory. However, an error was reported while trying to read the file

**System action:** The JVM is terminated.

**User response:** If the problem remains, contact your IBM service representative.

**UTE008 Out of memory in rasTraceRegister**

**Explanation:** While registering an external trace listener, the JVM attempted to malloc memory to hold the listener's address, but the malloc failed

**System action:** `JNI_ENOMEM` is returned to the calling program.

**User response:** Try running the JVM with a larger maximum heap size (by using the **-Xmx** option). If the problem remains, contact your IBM service representative

**UTE009 Invalid module number (%d) for %**

**Explanation:** Internal error in trace initialization.

**System action:** Tracing will not be enabled for this executable.

**User response:** Contact your IBM service representative.

**UTE010 Name mismatch for module number %d; is %s, should be %s**

**Explanation:** Internal error in trace initialization.

**System action:** Tracing will not be enabled for this executable.

**User response:** Contact your IBM service representative.

**UTE011 Active tracepoint array length for %s is %d; should be %d**

**Explanation:** Internal error in trace initialization.

**System action:** Tracing will not be enabled for this executable.

**User response:** Contact your IBM service representative.

**UTE012 Trace configuration mismatch for %s**

**Explanation:** The CRC for the named executable does not match the previously stored value.

**System action:** Trace is not enabled for this executable.

**User response:** Contact your IBM service representative.

**UTE013 Invalid module number (%d) for %s**

**Explanation:** Internal error in trace termination.

**System action:** Tracing will not be terminated for this executable.

**User response:** Contact your IBM service representative.

**UTE014 Name mismatch for module number %d; is %s, should be %s**

**Explanation:** Internal error in trace termination.

**System action:** Tracing will not be terminated for this executable.

**User response:** Contact your IBM service representative.

**UTE016 utcMemAlloc failure in utsTraceSet**

**Explanation:** While trying to process a trace option, a malloc failed.

**System action:** The JVM may fail as a result of this error.

**User response:** System memory (not Java heap) is full. Consider reducing the size of the Java heap to save space.

**UTE017 Unable to purge trace buffer for thread %p**

**Explanation:** While external trace (trace to a file) was running, trace records were lost. When the buffers were flushed at the point a signal was received, this buffer could not be written.

**System action:** The indicated trace buffer will not be written to disk.

**User response:** Be aware when processing trace files that, because of this error, the files might not be complete. If the problem remains, contact your IBM service representative.

**UTE018 %d trace records lost**

**Explanation:** Trace records are lost when the trace buffers option (**-Dibm.dg.trc.buffers**) is set to *nodynamic*, both buffers are full, and the first buffer has not yet been written to disk when the second buffer fills up. The JVM wants to switch back to tracing into the first buffer, but because it has not been written to disk, new records are lost. When the first buffer is written, it is reused for new trace records, but this warning tells you how many records were lost before that happened.

**System action:** None.

**User response:** When specifying the buffers property, specify *dynamic* or remove the specification of *nodynamic* (for example, **-Dibm.dg.trc.buffers=16k,dynamic**).

**UTE019 Unable to obtain storage for thread control block**

**Explanation:** While processing the startup of a new thread, no storage was available to allocate a trace thread control block.

**System action:** Memory for this process is either fragmented or exhausted. If possible the JVM will continue to run, but may fail as a result of this condition.

**User response:** If running with a very large Java heap, try reducing its size to allow allocation of non-Java objects.

**UTE020 Unable to obtain storage for excluded command list**

**Explanation:** While processing trace initialization, no storage was available to allocate an internal structure.

**System action:** Memory for this process is either fragmented or exhausted. If possible the JVM will continue to run, but may fail as a result of this condition.

**User response:** If running with a very large Java heap, try reducing its size to allow allocation of non-Java objects.

**UTE021 Unable to obtain storage for excluded command**

**Explanation:** While processing trace initialization, no storage was available to allocate an internal structure.

**System action:** Memory for this process is either fragmented or exhausted. If possible the JVM will continue to run, but may fail as a result of this condition.

**User response:** If running with a very large Java heap, try reducing its size to allow allocation of non-Java objects.

**UTE022 Initialization of traceLock failed**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**UTE023 Initialization of writeEvent failed**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**UTE024 Initialization of traceTerminated semaphore failed**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**UTE025 Initialization of writeInititialized semaphore failed**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**UTE026 Unable to obtain storage for global control block**

**Explanation:** While processing trace initialization, no storage was available to allocate an internal structure.

**System action:** Memory for this process is either fragmented or exhausted. If possible the JVM will continue to run, but may fail as a result of this condition.

**User response:** If running with a very large Java heap, try reducing its size to allow allocation of non-Java objects.

**UTE027 Error processing early options**

**Explanation:** An invalid trace option was detected during trace initialization.

**System action:** JVM initialization may fail due to this error.

**User response:** Check the syntax of any trace options specified in the JVM startup options and system properties. If the options are correct, contact your IBM service representative.

**UTE028 Error processing control file**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**UTE029 Error initializing module blocks**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**UTE030 Error processing options**

**Explanation:** An invalid trace option was detected during trace initialization.

**System action:** JVM initialization may fail due to this error.

**User response:** Check the syntax of any trace options specified in the JVM startup options and system properties. If the options are correct, contact your IBM service representative.

**UTE031 AddComponent failed to allocate memory for %d applids**

**Explanation:** While processing trace initialization, no storage was available to allocate an internal structure.

**System action:** Memory for this process is either fragmented or exhausted. If possible the JVM will continue to run, but may fail as a result of this condition.

**User response:** If running with a very large Java heap, try reducing its size to allow allocation of non-Java objects.

**UTE032 AddComponent failed to allocate memory for applid**

**Explanation:** While processing trace initialization, no storage was available to allocate an internal structure.

**System action:** Memory for this process is either fragmented or exhausted. If possible the JVM will continue to run, but may fail as a result of this condition.

**User response:** If running with a very large Java heap, try reducing its size to allow allocation of non-Java objects.

**UTE033 Out of memory handling applids**

**Explanation:** While processing trace initialization, no storage was available to allocate an internal structure.

**System action:** Memory for this process is either fragmented or exhausted. If possible the JVM will continue to run, but may fail as a result of this condition.

**User response:** If running with a very large Java heap, try reducing its size to allow allocation of non-Java objects.

**UTE101 RC %d from utcEventWait in waitEvent**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**UTE102 RC %d from utcEventPost in postEvent**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**UTE103 Out of memory in initTraceHeader**

**Explanation:** While processing trace initialization, no storage was available to allocate an internal structure.

**System action:** Memory for this process is either fragmented or exhausted. If possible the JVM will continue to run, but may fail as a result of this condition.

**User response:** If running with a very large Java heap, try reducing its size to allow allocation of non-Java objects.

**UTE104 Error opening tracefile: %s**

**Explanation:** The JVM tried to open a trace output file (name supplied in message), but the open failed.

**System action:** Various.

**User response:** Check whether your problem with opening the file is caused by, for example, disk space or security settings. If the problem remains, contact your IBM service representative.

**UTE105 Error writing header to tracefile: %s**

**Explanation:** The JVM attempted to write the trace file header to disk but the operation was unsuccessful or only partially successful.

**System action:** Various.

**User response:** Check whether your problem with opening the file is caused by, for example, disk space or security settings. If the problem remains, contact your IBM service representative.

**UTE106 Error from utcFileSetLength for tracefile: %s**

**Explanation:** The JVM attempted to write a trace buffer to disk, but the operation was unsuccessful or only partially successful.

**System action:** None.

**User response:** Check whether your problem with opening the file is caused by, for example, disk space or security settings. If the problem remains, contact your IBM service representative.

**UTE107 Error writing to tracefile: %s**

**Explanation:** The JVM attempted to write a trace buffer to disk, but the operation was unsuccessful or only partially successful.

**System action:** None.

**User response:** Check whether your problem with opening the file is caused by, for example, disk space or security settings. If the problem remains, contact your IBM service representative.

**UTE108 Error opening next generation: %s**

**Explanation:** The trace is moving on to the next generation file and so needs to open the file that is suffixed for this new generation. Unfortunately it cannot open the new file.

**System action:** When running with the PD build, the JVM terminates with an assertion failure.

**User response:** Check whether your problem with opening the file is caused by, for example, disk space or security settings. If the problem remains, contact your IBM service representative.

**UTE109 Error performing seek in tracefile: %s**

**Explanation:** The JVM has attempted to skip past the header of a trace file so that it can write more trace data into it. Unfortunately, this failed.

**System action:** When running with the PD build, the JVM terminates with an assertion failure.

**User response:** Check whether your problem with opening the file is caused by, for example, disk space or security settings. If the problem remains, contact your IBM service representative.

**UTE110 Error performing seek in tracefile: %s**

**Explanation:** The JVM has attempted to skip past the header of a trace file so that it can write more trace data into it. Unfortunately, this failed.

**System action:** When running with the PD build, the JVM terminates with an assertion failure.

**User response:** Check whether your problem with opening the file is caused by, for example, disk space or security settings. If the problem remains, contact your IBM service representative.

**UTE111 Error opening next state file: %s**

**Explanation:** When filling one state trace file and switching to the other, an error occurred while the new (named) file was being opened.

**System action:** When running with the PD build, the JVM terminates with an assertion failure.

**User response:** Consider enabling fewer state trace tracepoints. If the problem remains, contact your IBM service representative.

**UTE112 Error performing seek in tracefile: %s**

**Explanation:** The JVM has attempted to skip past the header of a trace file so that it can write more trace data into it. Unfortunately, this failed.

**System action:** When running with the PD build, the JVM terminates with an assertion failure.

**User response:** Check whether your problem with opening the file is caused by, for example, disk space or security settings. If the problem remains, contact your IBM service representative.

**UTE113 RC %d from utsThreadStop in traceWrite**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**UTE114 RC %d from utsThreadStop in traceWrite**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**UTE115 At least one trace record lost**

**Explanation:** Trace Buffers are set to *nodynamic* and both buffers are full. Before the first buffer could be written to disk, the second one has filled up and the system now wants to write trace records to the first buffer again. However, it cannot do this without overwriting trace data that is already present.

**System action:** All new trace data is discarded until the first buffer can be written to disk.

**User response:** Specify the keyword dynamic on the buffers option (for example, **-Dibm.dg.trc.buffers=16k,dynamic**), or remove the *nodynamic* keyword that is there.

**UTE116 Out of memory obtaining trace buffer**

**Explanation:** An attempt to obtain an additional trace buffer failed; no storage was available.

**System action:** Trace data will be lost for the thread that encountered this error. If possible, the JVM will continue to run, but may fail as a result of this condition.

**User response:** If running with a very large Java heap, try reducing its size to allow allocation of non-Java objects. Also, consider reducing the volume of data being traced, or moving the output file to a faster medium, or both.

**UTE117 Counter wrap for tracepoint %6.6X**

**Explanation:** When you were using the trace count option (**-Dibm.dg.trc.count**), a tracepoint was incremented so many times that the count has wrapped back to zero.

**System action:** The counter for the specified tracepoint has wrapped to zero and will count upwards again from there.

**User response:** Determine the action to take.

**UTE201 utcMemAlloc failure in addTraceCmd**

**Explanation:** While trying to process a trace option, a malloc failed.

**System action:** The JVM may fail as a result of this error.

**User response:** System memory (not Java heap) is full. Consider reducing the size of the Java heap to save space.

**UTE202 Invalid multiplier specified for buffer size**

**Explanation:** Buffer size can be specified in multiples of KB or MB. For example, *buffers=16k*. Note that only lowercase *k* or *m* is used. All other letters in this position are not valid.

**System action:** The JVM is terminated.

**User response:** Try again, this time specifying *k* or *m*.

**UTE203 Length of buffer size parameter invalid**

**Explanation:** A buffer size must be between two and five characters in length including the *k* or *m*. The one that you specified was too short or too long.

**System action:** The JVM is terminated.

**User response:** Try again, this time specifying a value between two and five characters in length for the buffer size (for example **-Dibm.dg.trc.buffers=1234k**).

**UTE204 Buffer size not specified**

**Explanation:** The **buffers** system property expects a buffer size to be specified. You did not provide one.

**System action:** The JVM is terminated.

**User response:** Try again, this time specifying a size for the buffer (for example, **-Dibm.dg.trc.buffers=8k**).

**UTE205 Dynamic or Nodynamic keyword expected**

**Explanation:** The buffers system property takes an optional second argument (after the size). The only allowed values for this are *dynamic* or *nodynamic*.

**System action:** The JVM is terminated.

**User response:** Try again, this time specifying a second argument of *dynamic* or *nodynamic* on the **buffers** property (for example, **-Dibm.dg.trc.buffers=16k,nodynamic**) or by omitting the argument entirely.

**UTE206 Unrecognized keyword in buffer specification**

**Explanation:** The **buffers** system property contains an unrecognized keyword.

**System action:** The JVM is terminated.

**User response:** Correct the syntax and try again.

**UTE207 Too many keywords in buffer specification**

**Explanation:** The buffers system property contains too many keywords.

**System action:** The JVM is terminated.

**User response:** Correct the syntax and try again

**UTE208 Usage: buffers=nnnn{k|m} [,dynamic|nodynamic]**

**Explanation:** This explanatory message describes the syntax for the buffers system property.

**System action:** The JVM is terminated.

**User response:** Correct the syntax and try again

**UTE209 Out of memory handling exception property**

**Explanation:** While processing trace initialization, no storage was available to allocate an internal structure.

**System action:** Memory for this process is either fragmented or exhausted. If possible, the JVM will continue to run, but may fail as a result of this condition.

**User response:** If running with a very large Java heap, try reducing its size to allow allocation of non-Java objects.

**UTE210 Filename not supplied in exception specification**

**Explanation:** When specifying the exception output file (**-Dibm.dg.trc.exception.output**), you did not provide a file name.

**System action:** The JVM is terminated.

**User response:** Try again, specifying a file name for the **exception.output** property (for example,-**Dibm.dg.trc.exception.output=fred.trc**).

**UTE211 Invalid multiplier for exception wrap limit**

**Explanation:** Wrap limit can be specified in multiples of KB (*k*) or MB (*m*); for example, **-Dibm.dg.trc.exception.output=except,2m**.

**System action:** The JVM is terminated.

**User response:** Try again, this time specifying *k* or *m*.

**UTE212 Length of wrap limit parameter invalid**

**Explanation:** A wrap limit must be between two and five characters in length including the *k* or *m*. The one that you specified was too short or too long.

**System action:** The JVM is terminated.

**User response:** Try again, this time specifying a value two through five characters long for the buffer size (for example, **-Dibm.dg.trc.exception.output=fred.trc,1234k**).

**UTE213 Too many keywords in exception specification**

**Explanation:** Only two allowable parameters can follow the **exception.output** parameter. They are filename and wrap limit. Because you tried to put more options there, this is an error.

**System action:** The JVM is terminated.

**User response:** Remove the third (and later) keywords that are following on the **exception.output** specification, then retry.

**UTE214 Usage: exception.output=filename[,nnnn{k|m}]**

**Explanation:** The way in which you have attempted to specify **exception.output** does not match the displayed usage.

**System action:** The JVM is terminated.

**User response:** Correct your specification of **exception.output** and retry.

---

**UTE215 Out of memory handling state.output property**

**Explanation:** While processing trace initialization, no storage was available to allocate an internal structure.

**System action:** Memory for this process is either fragmented or exhausted. If possible, the JVM will continue to run, but may fail as a result of this condition.

**User response:** If running with a very large Java heap, try reducing its size to allow allocation of non-Java objects.

---

**UTE216 Filename not supplied in state.output specification**

**Explanation:** State trace output requires a filename, but none was supplied.

**System action:** The JVM terminates.

**User response:** When you specify the state trace output, ensure that you provide a filename (for example, **-Dibm.dg.trc.state.output=fred.trc**).

---

**UTE217 Invalid multiplier for state wrap limit**

**Explanation:** State trace output file wrap size can be specified in multiples of KB (*k*) or MB (*m*) only.

**System action:** The JVM terminates.

**User response:** Retry, specifying state trace output file wrap size in multiples of KB (*k*) or MB (*m*); for example, **state.output=filename.trc,2m**.

---

**UTE218 Length of wrap limit parameter invalid**

**Explanation:** The length of the state trace wrap limit must be two through five characters (including the multiplier *k* or *m*, if specified). This rule was broken.

**System action:** The JVM terminates.

**User response:** Retry, specifying state trace output file wrap size in two through five characters including the multiplier (for example, **-Dibm.dg.trc.state.output=filename.trc,1234k**).

---

**UTE219 Too many keywords in state.output specification**

**Explanation:** **State.output** is followed by a maximum of two parameters: the file name and the file wrap size. A third parameter is not valid.

**System action:** The JVM terminates.

**User response:** Remove the invalid third (and later) parameters on the state.output specification, and retry.

---

**UTE220 Usage: state.output=filename[,nnnn{k|m}]**

**Explanation:** The usage for a **state.output** parameter is as shown.

**System action:** The JVM terminates.

**User response:** Correct your input to match the usage displayed and retry.

---

**UTE221 Out of memory handling output property**

**Explanation:** While processing trace initialization, no storage was available to allocate an internal structure.

**System action:** Memory for this process is either fragmented or exhausted. If possible the JVM will continue to run, but may fail as a result of this condition.

**User response:** If running with a very large Java heap, try reducing its size to allow allocation of non-Java objects.

---

**UTE222 Filename not supplied in output specification**

**Explanation:** State trace output requires a filename, but none was supplied.

**System action:** The JVM terminates.

**User response:** When you specify the state trace output, ensure that you provide a filename (for example, **-Dibm.dg.trc.state.output=fred.trc**).

---

**UTE223 Invalid multiplier for trace wrap limit**

**Explanation:** Output file wrap size can be specified in multiples of KB or MB; for example, **output=filename.trc,2m**. Note that only lowercase *k* or *m* is used. All other letters in this position are not valid.

**System action:** The JVM is terminated.

**User response:** Try again, this time specifying *k* or *m*.

---

**UTE224 Length of wrap limit parameter invalid**

**Explanation:** A wrap limit must be between two and five characters in length including the *k* or *m*. The one that you specified was too short or too long.

**System action:** The JVM is terminated.

**User response:** Try again, this time specifying a value two through five characters long for the buffer size (for example, **-Dibm.dg.trc.exception.output=fred.trc,1234k**).

---

**UTE225 Invalid number of trace generations**

**Explanation:** When specified, the number trace generations must be 2 through 36. The number that you specified falls outside these limits.

**System action:** The JVM is terminated.

**User response:** Try again, this time specifying a value 2 through 36 for the number of trace generations (for example, **-Dibm.dg.trc.output=trace.out,2m,10**).

**UTE226 Invalid filename for generation mode**

**Explanation:** When trace generations are specified, the trace output file name must contain a **#** character. This is replaced with the generation character in each trace generation file. The name that you specified does not contain a **#**.

**System action:** The JVM is terminated.

**User response:** Try again, this time specifying a filename that contains a **#** (for example, **-Dibm.dg.trc.output=trace#.out,2m,10**).

**UTE227 Length of generation parameter invalid**

**Explanation:** Because the number of trace generations must be 2 through 36, it makes no sense for the length of the trace generations parameter to be anything other than 1 or 2 characters long. The argument that you supplied was less than 1, or greater than 2 characters long.

**System action:** The JVM is terminated.

**User response:** Try again, this time specifying a value 2 through 36 for the number of trace generations (for example, **-Dibm.dg.trc.output=trace.out,2m,10**).

**UTE228 Too many keywords in output specification**

**Explanation:** The output specification supplies an output filename, (optionally) an output file wrap size, and (optionally) several generations. Further arguments are meaningless and should not be specified.

**System action:** The JVM is terminated.

**User response:** Try again, this time omitting the fourth (and following) meaningless arguments.

**UTE229 Usage: output=filename[,nnnn{k|m}[,n]]**

**Explanation:** The output specification that you supplied does not meet the described usage.

**System action:** The JVM is terminated.

**User response:** Correct the output specification so that it meets the described usage, and try again.

**UTE230 Empty clauses not allowed in trigger property.**

**Explanation:** A null clause was found in a trigger property.

**System action:** The JVM fails to initialize.

**User response:** Correct the trigger property, then retry the operation. This error probably occurred because you entered something like **trigger=method,,tpid** (that is, you entered too many commas).

**UTE231 utcMemAlloc failure for FormatSpecPath**

**Explanation:** While trying to process a trace option, a malloc failed.

**System action:** The JVM may fail as a result of this error.

**User response:** System memory (not Java heap) is full. Consider reducing the size of the Java heap to save space.

**UTE232 utcMemAlloc failure for Suffix**

**Explanation:** While trying to process a trace option, a malloc failed.

**System action:** The JVM may fail as a result of this error.

**User response:** System memory (not Java heap) is full. Consider reducing the size of the Java heap to save space.

**UTE233 utcMemAlloc failure for traceControlPath**

**Explanation:** While trying to process a trace option, a malloc failed.

**System action:** The JVM may fail as a result of this error.

**User response:** System memory (not Java heap) is full. Consider reducing the size of the Java heap to save space.

**UTE234 Trace control already loaded**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**UTE235 Incorrect TRACECONTROL value**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**UTE236 LIBPATH and TRACECONTROL options are mutually exclusive**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**UTE237 resumecount takes a single integer value from -99999 to +99999**

**Explanation:** The **ibm.dg.trc.resumecount** property is an integer value -99999 through +99999. The value that you specified was not in this range.

**System action:** The JVM fails to initialize.

**User response:** Correct the **resumecount** property, then retry the operation.

**UTE238 suspendcount takes a single integer value from -99999 to +99999**

**Explanation:** The **ibm.dg.trc.suspendcount** property is an integer value -99999 through +99999. The value that you specified was not in this range.

**System action:** The JVM fails to initialize.

**User response:** Correct the **suspendcount** property, then retry the operation.

**UTE239 resumecount and suspendcount may not both be set.**

**Explanation:** You attempted to set the resumecount and suspendcount properties at the same time. This is not allowed.

**System action:** The JVM fails to initialize.

**User response:** Decide which property you want to use, then remove the other

**UTE240 Out of memory while processing properties file**

**Explanation:** While processing trace initialization, no storage was available to allocate an internal structure.

**System action:** Memory for this process is either fragmented or exhausted. If possible the JVM will continue to run, but may fail as a result of this condition.

**User response:** If running with a very large Java heap, try reducing its size to allow allocation of non-Java objects.

**UTE241 Unable to open properties file %s**

**Explanation:** The JVM was unable to open the properties file that was listed in the message.

**System action:** The JVM is terminated.

**User response:** Ensure that the properties file that you have specified really exists. If the problem remains, contact your IBM service representative.

**UTE242 Unable to determine size of properties file %s**

**Explanation:** Having opened the trace properties file mentioned in the message, the JVM was unable to determine its size.

**System action:** The JVM is terminated.

**User response:** Ensure that the file that you specified is a valid properties file, and that it is readable. If the problem remains, contact your IBM service representative.

**UTE243 Cannot obtain memory to process %s**

**Explanation:** While processing trace initialization, no storage was available to allocate an internal structure.

**System action:** Memory for this process is either fragmented or exhausted. If possible the JVM will continue to run, but may fail as a result of this condition.

**User response:** If running with a very large Java heap, try reducing its size to allow allocation of non-Java objects.

**UTE245 Error reading properties file %s**

**Explanation:** To process the trace properties file, it is read into memory. Unfortunately, the call to read it into memory has failed.

**System action:** The JVM is terminated.

**User response:** Contact your IBM service representative.

**UTE246 utcMemAlloc failure for FormatSpec**

**Explanation:** While trying to process a trace option, a malloc failed.

**System action:** The JVM may fail as a result of this error.

**User response:** System memory (not Java heap) is full. Consider reducing the size of the Java heap to save space.

**UTE247 Unrecognized line in %s: ″%s″**

**Explanation:** While reading the trace properties file, a line has been found that contains a keyword that is not recognized. The properties file name and the offending line are included in the text of the message.

**System action:** The JVM is terminated.

**User response:** Correct the line in error and try again.

**UTE248 Unrecognized option : ″%s″**

**Explanation:** While processing the trace options, an unrecognized keyword has been encountered.

**System action:** The JVM is terminated.

**User response:** Correct or remove the option in error and try again.

**UTE249 utcMemAlloc failure for UtSpecial**

**Explanation:** While trying to process a trace option, a malloc failed.

**System action:** The JVM may fail as a result of this error.

**User response:** System memory (not Java heap) is full. Consider reducing the size of the Java heap to save space.

**UTE250 utcMemAlloc failure for UtSpecial**

**Explanation:** While trying to process a trace option, a malloc failed.

**System action:** The JVM may fail as a result of this error.

**User response:** System memory (not Java heap) is full. Consider reducing the size of the Java heap to save space.

**UTE251 utcMemAlloc failure for UtItem**

**Explanation:** While trying to process a trace option, a malloc failed.

**System action:** The JVM may fail as a result of this error.

**User response:** System memory (not Java heap) is full. Consider reducing the size of the Java heap to save space.

**UTE252 utcMemAlloc for trace array for %s failed**

**Explanation:** While trying to process a trace option, a malloc failed.

**System action:** The JVM may fail as a result of this error.

**User response:** System memory (not Java heap) is full. Consider reducing the size of the Java heap to save space.

**UTE253 Invalid range: %6.6X-%6.6X**

**Explanation:** You can specify ranges of tracepoints; for example, **tpid(c001-c01f)**. However if you do, the second number must be bigger than the first.

**System action:** The JVM is terminated.

**User response:** Correct the tpid range and retry.

**UTE254 Tracepoint id is not a valid hex string**

**Explanation:** Tpids (trace point ids) are expressed as a hex number 1 through 6 characters long and including only the characters 0 through 9 and a through f (also A through F). The tpid that you specified does not meet these criteria.

**System action:** The JVM is terminated.

**User response:** Correct the tpid and try again.

**UTE255 Invalid range: %6.6X-%6.6X**

**Explanation:** Tpids (trace point ids) are expressed as a hex number 1 through 6 characters long and including only the characters 0 through 9 and a through f (also A through F). The tpid that you specified does not meet these criteria.

**System action:** The JVM is terminated.

**User response:** Correct the tpid and try again.

**UTE256 Invalid tracepoint id: %6.6X**

**Explanation:** The specified tracepoint does not exist in this build.

**System action:** The JVM may terminate.

**User response:** Modify the trace properties, removing or correcting the reference to this tpid.

**UTE257 Tracepoint %6.6X is not included in this build**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**UTE258 Tracepoint id is not a valid hex string**

**Explanation:** You used an invalid hex number when specifying tracepoint ids for application trace

**System action:** The JVM terminates.

**User response:** Correct the specification and retry.

**UTE259 utcMemAlloc failure in setTraceState**

**Explanation:** While trying to process a trace option, a malloc failed.

**System action:** The JVM may fail as a result of this error.

**User response:** System memory (not Java heap) is full. Consider reducing the size of the Java heap to save space.

**UTE260 Trace selection specification incomplete:\n%s**

**Explanation:** The input line shown above ends at an unexpected point.

**System action:** The JVM is terminated.

**User response:** Correct the line that is indicated in the message and try again.

**UTE261 Syntax error encountered at offset %d in:\n%s**

**Explanation:** A syntax error is at the indicated position in the line displayed.

**System action:** The JVM is terminated.

**User response:** Correct the line that is indicated in the message, and try again.

**UTE262 Error processing options**

**Explanation:** An invalid trace option was detected during trace initialization.

**System action:** JVM initialization may fail due to this error.

**User response:** Check the syntax of any trace options specified in the JVM start-up options and system properties. If the options are correct, contact your IBM service representative.

**UTE263 Error processing options**

**Explanation:** An invalid trace option was detected during trace initialization.

**System action:** JVM initialization may fail due to this error.

**User response:** Check the syntax of any trace options specified in the JVM startup options and system properties. If the options are correct, contact your IBM service representative.

**UTE301 RC %d from utcMutexEnter in getTraceLock**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**UTE302 RC %d from utcMutexExit in freeTraceLock**

**Explanation:** Internal error.

**System action:** None.

**User response:** Contact your IBM service representative.

**UTE303 Invalid special character '%c' in a trace filename. Only %p, %d and %t are allowed.**

**Explanation:** Trace output file naming can be modified using special operators. The only supported operators are %p, %d and %t (process ID, date, and time, respectively).

**System action:** The JVM fails to initialize.

**User response:** Remove any % characters from the filename and retry.

**UTE304 Missing closing bracket(s) in trigger property.**

**Explanation:** The **trigger** property did not end in a closing bracket.

**System action:** The JVM fails to initialize.

**User response:** Correct the brackets on the trigger property, then retry the operation.

**UTE305 Out of memory processing trigger property.**

**Explanation:** While processing trace initialization, no storage was available to allocate an internal structure.

**System action:** Memory for this process is either fragmented or exhausted. If possible the JVM will continue to run, but may fail as a result of this condition.

**User response:** If running with a very large Java heap, try reducing its size to allow allocation of non-Java objects.

**UTE306 TraceFormat.dat is incorrect version**

**Explanation:** The Trace format file, TraceFormat.dat, contains a version field. During trace initialization, the contents of this field have been checked and are not what the JVM expected. This could mean that you are picking up the Trace format file from a different release of the JVM.

**System action:** Warning only. This message is issued, but no action is taken.

**User response:** Specify the location of the correct version of the Trace Format file by using the **-Dibm.dg.trc.format=<filename>** system property.

**UTE307 TraceFormat.dat is incorrect format**

**Explanation:** The Trace format file TraceFormat.dat contains a version field. During trace initialization, the JVM could not locate this version number because the file is in an unexpected format.

**System action:** The JVM terminates.

**User response:** Specify the location of the correct version of the Trace Format file by using the **-Dibm.dg.trc.format=<filename>** system property.

**UTE308 TraceFormat.dat is incorrect version**

**Explanation:** The Trace format file TraceFormat.dat contains a version field. During trace initialization, the contents of this field have been checked and were not what this JVM expected. This could mean that you are picking up the Trace format file from a different release of the JVM.

**System action:** Warning only. This message is issued but no action is taken.

**User response:** Specify the location of the correct version of the Trace Format file by using the **-Dibm.dg.trc.format=<filename>** system property.

**UTE309 Cannot obtain memory for format table**

**Explanation:** While processing trace initialization, no storage was available to allocate an internal structure.

**System action:** Memory for this process is either fragmented or exhausted. If possible the JVM will continue to run, but may fail as a result of this condition.

**User response:** If running with a very large Java heap, try reducing its size to allow allocation of non-Java objects.

**UTE310 Unable to open trace format file %s**

**Explanation:** During trace initialization, the JVM reads the Trace format file into memory. It was unable to open the file, so was unable to do this.

**System action:** Trace format file is not read. Trace is not initialized.

**User response:** Specify the location of the correct version of the Trace format file by using the **-Dibm.dg.trc.format=<filename>** system property. If the problem remains, contact your IBM service representative.

**UTE311 Unable to determine size of trace format file %s**

**Explanation:** During trace initialization, the JVM reads the Trace format file into memory. It was unable to open the file, so was unable to do this.

**System action:** Trace format file is not read. Trace is not initialized.

**User response:** Specify the location of the correct version of the Trace Format file by using the **-Dibm.dg.trc.format=<filename>** system property. If the problem remains, contact your IBM service representative.

**UTE312 Cannot obtain memory to process %s**

**Explanation:** While processing trace initialization, no storage was available to allocate an internal structure.

**System action:** Memory for this process is either fragmented or exhausted. If possible the JVM will continue to run, but may fail as a result of this condition.

**User response:** If running with a very large Java heap, try reducing its size to allow allocation of non-Java objects.

**UTE313 Error reading trace format file %s**

**Explanation:** During trace initialization, the JVM has attempted to create an in-memory copy of the Trace format file. An error occurred while the file was being read.

**System action:** The JVM terminates.

**User response:** Specify the location of the correct version of the Trace format file by using the **-Dibm.dg.trc.format=<filename>** system property. If the problem remains, contact your IBM service representative.

**UTE314 Unable to open tracepoint counter file**

**Explanation:** The tracepoint count option has been specified and an attempt was made to open the count output file, but this failed.

**System action:** The JVM continues.

**User response:** If the problem remains, contact your IBM service representative.

**UTE315 Counters redirected to stderr**

**Explanation:** The tracepoint count option has been specified but the attempt to open the count output file failed. The output is being redirected to stderr instead

**System action:** The JVM continues.

**User response:** None.

**UTE316 Signed number not permitted in this context ″%s″**

**Explanation:** When a clause in the system property **-Dibm.dg.trc.trigger** was being processed, a negative delay count was found.

**System action:** The JVM fails to initialize.

**User response:** Check the contents of the **trigger=tpid(...)**, **method(...)**, and **group(...)** clauses. If a **delaycount** is specified, it must be a positive number.

**UTE317 Invalid character(s) encountered in decimal number ″%s″**

**Explanation:** When a clause in the system property **-Dibm.dg.trc.trigger** was being processed, a non-numeric character was found.

**System action:** The JVM fails to initialize.

**User response:** Check the contents of the **trigger=tpid(...)**, **method(...)**, and **group(...)** clauses. If a **delaycount** is specified, it must contain only the characters 0 through 9.

**UTE318 Number too long or too short ″%s″**

**Explanation:** When a clause in the system property **-Dibm.dg.trc.trigger** was being processed, a bad delay count was found.

**System action:** The JVM fails to initialize.

**User response:** Check the contents of the **trigger=tpid(...)**, **method(...)**, and **group(...)** clauses. If a **delaycount** is specified, it must be between one and eight digits long.

**UTE319 Cannot allocate memory for trace module blocks**

**Explanation:** While processing trace initialization, no storage was available to allocate an internal structure.

**System action:** Memory for this process is either fragmented or exhausted. If possible the JVM will continue to run, but may fail as a result of this condition.

**User response:** If running with a very large Java heap, try reducing its size to allow allocation of non-Java objects.

# Appendix G. Command-line parameters

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:



## General command-line parameters

**-cp, -classpath<directories and zip or jar files separated by ;> (or : on Unix)**
Sets search path for application classes and resources.

**-help, -?**
Prints a usage message.

**-showversion**
Prints product version and continues.

**-verbose[:class | gc | jni]**
Enables verbose output.

**-verbose:Xclassdep**
Traces all the class loading and the method and classnames with line numbers.

**-version**
Prints product version.

## System property command-line parameters

**-D<name>=<value>**
Sets a system property.

**-Dcom.ibm.cacheLocalHost=true**
Multiple calls to the java.net.InetAddress.getLocalHost() can impact JVM performance. Set this property to enable caching of the local host name.

**-Dibm.ci.verbose**
This system property traces the initialization routines and is useful for debugging any problems occurring during JVM creation.

**-Dibm.cl.eagerresolution**
Requires no value to turn on eager class resolution

**-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.jvm.events.output={stderr | filename}, -Dibm.jvm.crossheap.events, -Dibm.jvm.resettrace.events, -Dibm.jvm.unresettable.events.level={min | max} (z/OS only)**
Setting these properties and running the PD build with JIT disabled produces more information whenever an Unresettable Event occurs. This information helps in debugging the problem. The output is redirected to STDERR if the value of ibm.jvm.events.output is set to stderr. This applies only for a resettable JVM.

**-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.

**-Dibm.xe.coe.name={exception}**
The value of the property is the package description of the exception. Setting this property generates a system dump when the specified exception occurs.

**-Djava.compiler={ NONE | jitc }**
Disable the Java compiler by setting to NONE. Enable JIT compilation by setting to jitc.

**-Djava.net.connectiontimeout={n}**
'n' is the number of seconds to wait for the connection to be established with the server. If this option is not specified in the command line, the default value of 0 (infinity) is used. The value can be used as a timeout limit when an asynchronous java-net application is trying to establish a connection with its server. If this value is not set, a java-net application waits until the default connection timeout value is met. For instance, `java -Djava.net.connectiontimeout=2 TestConnect` causes the java-net client application to wait only 2 seconds to establish a connection with its server.

**-Djava.net.preferIPv4Stack={true | false}**
When set to **true**, the Java Virtual Machine uses only the IPv4 stack, and will not be able to communicate with IPv6 hosts. By default, this parameter is set to false.

**-Djava.net.preferIPv6Address={true | false}**
When set to **true**, the Java Virtual Machine will connect using IPv6 in preference to IPv4. By default, this parameter is set to false.

**-Djava.rmi.server.logCalls={true | false}**
If this property is **true**, incoming calls and exceptions thrown from incoming calls will be logged to System.err.

**-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 same for Swing programs running with the Windows Look and Feel.

## Nonstandard command-line parameters

The **-X** parameters are nonstandard and subject to change without notice.

Parameters that relate to the Garbage Collector are listed under "Garbage Collector command-line parameters" on page 491.

**-X**
Prints help on nonstandard options.

**-Xbootclasspath:<directories and zip or jar files separated by ;> (or : on Unix)**
Sets search path for bootstrap classes and resources.

**-Xbootclasspath/a:<directories and zip or jar files separated by ;> (or : on Unix)**
Appends to the bootstrap class path.

**-Xbootclasspath/p:<directories and zip or jar files separated by ;> (or : on Unix)**
Prepends to the bootstrap class path.

**-Xcheck:jni**
Performs additional checks for JNI functions.

**-Xcheck:nabounds**
Performs additional checks for JNI array operations.

**-Xdebug**
Starts the JVM with the debugger enabled. Used with **-Xrunjdwp**. On AIX PPC32 and PPC64 and Linux PPC32 and PPC64, from Service Refresh 5, this option starts the JVM using the alternative debug environment, described in Appendix I, "Using the alternative JVM for Java debugging," on page 499.

**-Xdisablejavadump**
Disables the Javadump facility.

**-Xfuture**
Enables strictest checks, anticipating future default.

**-Xifa:<on | off | force | projectn> (z/OS only)**
z/OS R6 provides the ability to run Java applications on a new type of special-purpose assist processors called the eServer zSeries Application Assist Processor (zAAP). The zSeries Application Assist Processor is also known as an IFA (Integrated Facility for Applications). The term IFA appears in panels, messages, and other z/OS information relating to the zSeries Application Assist Processor, including this publication.

The **-Xifa** setting is assumed to be **on** by default. It enables Java work to be run on IFAs if these are available. Only Java code and system native methods may be on IFA processors; this is achieved by requesting a switch to an IFA for qualifying work and a switch from IFA to a general-purpose processor when non-qualifying work is encountered.

Setting **off** disables the use of IFA processors. When **-Xifa:off** is specified, no other **-Xifa** options will be honored.

Setting **force** causes Java to continue calling the switch service, even when no IFA processors are available. You would typically set **force** to collect RMF data to assess potential IFA processor use. This option will have an affect only on version 1.6 or later of z/OS.

Setting **projectn** calculates projected IFA usage and prints this information to standard error at intervals no shorter than n minutes. When **project** without a time value is specified, the default is 5 minutes (**project5**). A value of **0** indicates that information is written only when Java terminates. The interval requested is not honored exactly; data is written whenever a potential switch to or from an IFA has been detected after the specified interval has elapsed. This option is honored on all versions of z/OS, but is primarily intended for assessing potential IFA processor use on versions below 1.6.

The **-Xifa** option can be specified multiple times and with combinations of options. For example:

```
-Xifa:on -Xifa:projectn
```

**-Xlp (AIX only)**
Requests the SDK to allocate the Java heap (the heap from which Java objects get allocated) with 16 MB large pages. For the 64 bit SDK, **-Xlp** requests also that mark and alloc bits are allocated with large pages. If large pages are not available, allocations use the standard 4 KB pages of AIX. AIX requires special configuration to enable large pages. For more information about how to configure AIX support for large pages, see http://www.ibm.com/servers/aix/whitepapers/large_page.html.

When the SDK attempts to allocate large page segments, it uses shmget() and shmatt() with the SHM_LGPG and SHM_PIN flags. As a result, this memory is allocated in mapped segments. This characteristic must be shown in any explicit setting of the **LDR_CNTRL=MAXDATA** environment variable. You might, however, no longer need to set **LDR_CNTRL=MAXDATA**, because the SDK automatically selects a value that allows for the required mappings. The **-Xlp** option replaces the environment variable **IBM_JAVA_LARGE_PAGE_SIZE**, which is now ignored if set.

**-Xnoagent**
Disables support for the oldjdb debugger.

**-Xoss<size>**
Sets maximum Java stack size for any thread (format = nn[K|M|G]).

**-Xpd**
Starts the Problem Determination build, as described in Appendix J, "Using a Problem Determination build of the JVM," on page 503, rather than the regular build.

**-Xquickstart**
Used for improving startup time of some Java applications. **-Xquickstart** causes 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.

**-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).

**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.

**-Xss<size>**
Sets maximum native stack size for any thread (format = nn[K|M|G]).

**-Xnosigchain**
Disables JVM signal handler chaining. The default is **-Xnosigchain** for z/OS, **-Xsigchain** for all other platforms.

**-Xsigchain**
Enables JVM signal handler chaining.

**-Xnosigcatch**
Disables JVM signal catching. The default is **-Xsigcatch**. This is useful for debugging problems in code that is protected by an IBM_HEAVYWEIGHT_TRY - CATCH block, where normally the problem is masked.

**-Xsigcatch**
Enables JVM signal catching.

## Garbage Collector command-line parameters

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 list contains all the Garbage Collector command-line parameters that are available in this release.

Note that reference to resettable JVM applies only to the z/OS operating system. Other platforms are not resettable and can never run in resettable mode.

**-verbosegc, -verbose:gc**
Prints garbage collection information. The format for the generated information is not architected and therefore varies from platform to platform and release to release.

**-Xcompactgc**
Compacts the heap every garbage collection cycle. The default is false (that is, the heap is not compacted). This is not recommended.

**-Xcompactexplicitgc**
Runs full compaction each time System.gc() is called. Its default behavior with a system.gc call is to perform a compaction only if an allocation failure triggered a garbage collection since the last system.gc call.

**-Xdisableexplicitgc**
Converts Java application call to java.lang.System.gc() into no-ops.

**-Xgcpolicy:<optthruput | optavgpause | subpool>**
Note that the *subpool* option was introduced in Version 1.4.1 Service Refresh 1 for AIX only.

Setting gcpolicy to *optthruput* disables concurrent mark. If you do not have pause time problems (as seen by erratic application response times), you should get the best throughput with this option. *Optthruput* is the default setting.

Setting gcpolicy 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 remove those problems at the cost of some throughput when running with the optavgpause option.

Setting gcpolicy to *subpool* enables improved object allocation that aims to achieve better performance in allocating objects on the heap. This setting might provide additional throughput optimization because it can improve the efficiency of object allocation and reduce lock contention on large SMP systems. Concurrent mark is disabled when this policy is enabled.

**-Xgcthreads<n>**
Sets the total number of threads that are used for garbage collection. On a system with n processors, the default setting for **-Xgcthreads** is *1* when the JVM is in resettable mode, and *n* when it is not in resettable mode.

**-Xinitacsh<size>**
Sets the initial size of the application-class system heap. This option is available only in the resettable JVM. Classes in this heap exist for the lifetime of the JVM. They are reset during a **ResetJavaVM()**, so are serially reusable by applications that are running in the JVM. Only one application-class system heap is present per Persistent Reusable JVM. In nonresettable mode, this option is ignored.

Example: **-Xinitacsh256k**

Default: 128K on 32-bit architecture, and 8M on 64-bit architecture.

**-Xinitsh<size>**
Sets the initial size of the system heap. Classes in this heap exist for the lifetime of the JVM. The system heap is never subjected to garbage collection. The maximum size of the system heap is unbounded.

Example: **-Xinitsh256k**

Default: 128K on 32-bit architecture, and 8M on 64-bit architecture.

**-Xinitth<size>**
Sets the initial size of the transient heap in the nonsystem heap. This option is available only in the resettable JVM. If this is not specified and **-Xms** is, the initial size is taken to be half the **-Xms** value. If **-Xms** is not specified, a value of half the platform-dependent default value is used.

Example: *-Xinitth2M*

Default: 1M÷2 = 512K

**-Xjvmset<size>**
Creates a master JVM. This option refers only to the resettable JVM. An optional size in MB can be specified to set the total size of the shared memory segment. The default is 1 MB. When **JNI_CreateJavaVM()** returns successfully,

the **extrainfo** field of the JavaVMOption contains the token that is to be passed to each worker. An attempt to create two master JVMs with the same token fails. The **-Xresettable** option must be used with this option when starting a master JVM.

**-Xjvmset**
This option relates only to the resettable JVM. It creates a worker JVM. The **extrainfo** field of the **JavaVMOption** must contain the token that is returned on the **-Xjvmset** option that was used to create the master JVM.

**-Xmaxdirectmemorysize<size>**
This option specifies the maximum amount of native memory allocated through DirectByteBuffer objects. System.gc is called after the limit is reached to clear unreferenced DirectByteBuffer objects from the Java heap.

**-Xmaxe<size>**
Specifies the maximum expansion size of the heap. The default is *0*. When the JVM is in resettable mode, this option sets the a maximum expansion size of <size>÷2 for the middleware and transient heaps.

**-Xmaxf<number>**
This is a floating point number between 0 and 1, which specifies the maximum percentage of free space in the heap. The default is *0.6*, or 60%. When this value is set to *0*, heap contraction is a constant activity. With a value of *1*, the heap never contracts. In resettable mode, this parameter applies to the middleware heap only.

**-Xmine<size>**
Specifies the minimum expansion size of the heap. The default is 1 MB. When the JVM is in resettable mode, this option sets a minimum expansion size of <size>÷2 for the middleware and transient heaps.

**-Xminf<number>**
This is a floating point number, 0 through 1, that specifies the minimum free heap size percentage. The heap grows if the free space is below the specified amount. In resettable mode, this option specifies the minimum percentage of free space for the middleware and transient heaps. The default is *.3* (that is 30%).

**-Xms<size>**
Sets the initial size of the heap. If this option is not specified, it defaults as follows:

Windows, AIX, and Linux: 4 MB.
OS/390: 1 MB

**-Xmx<size>**
Sets the maximum size of the heap. When the JVM is in resettable mode, this option sets the maximum size of the combined middleware and transient heaps. The middleware heap grows from the bottom of this region, and the transient heap grows from the top of the region. If this option is not specified, it defaults as follows:

- Windows: Half the real storage, but not less than 16 MB or more than 2 GB.
- OS/390 and AIX: 64 MB.
- Linux: Half the real storage, but not less than 16 MB or more than 512 MB.

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.

**-Xnoclassgc**
Disables class garbage collection.

**-Xnocompactgc**
Never compact the heap. Default is *false*.

**-Xnocompactexplicitgc**
Never runs compaction when System.gc() is called. Its default behavior with a system.gc call is to perform a compaction only if an allocation failure triggered a garbage collection since the last system.gc call.

**-Xresettable**
Specifies that this instance of the JVM can support the resettable JVM. Applies to the z/OS platform only.

**-Xtlhs**
Controls the minimum Thread Local Heap (TLH) size when **-Xgcpolicy:subpool** is specified . It is ignored for other gc policies. Possible values are from 512 to 8 KB. The default minimum TLH size is *768*. For example:

```
-Xtlhs4k
-Xtlhs800
```

**-Xverbosegclog:<path to file><filename>**
Causes verboseGC output to be written to the specified file. If the file cannot be found, verboseGC 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.

**Note:** The environment variable **IBM_JVMST_VERBOSEGC_LOG** has been removed from V1.4.1 and above.

**-Xverbosegclog:<path to file><filename, X, Y>**
Filename must contain a "#" (hash symbol), which is substituted with a generation identifier, starting at 1. X and Y are integers. This option works similarly to **-Xverbosegclog:<path to file><filename>**, but, in addition, the verboseGC output is redirected to X files, each containing verboseGC output from Y GC cycles.

**Note:** The environment variable **IBM_JVMST_VERBOSEGC_LOG** has been removed from 1.4.1 and above.

# Appendix H. 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 or the **IBM_JAVA_OPTIONS** environment variable
- **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

*Table 34. Cross platform defaults*

| JVM setting | Default | Setting affected by |
|---|---|---|
| Javadumps | Enabled | e |
| Javadumps on out of memory | Enabled | e |
| Heapdumps | Disabled | e |
| Heapdumps on out of memory | Enabled | e |
| Coredumps (not z/OS) | Disabled | e |
| Coredumps (z/OS only) | Enabled | e |
| Where Javadump and Coredump files appear | Current directory | e |
| Verbose output | Disabled | c |
| Boot classpath search | Disabled | c |
| JNI function checks | Disabled | c |
| JNI bound checks | Disabled | c |
| Remote debugging | Disabled | c |
| Strict conformancy checks | Disabled | c |
| Default thread stack size | 400 KB | c |
| Quickstart | Disabled | c |
| Reduced signalling | Disabled | c |
| Signal handler chaining | Enabled | c |
| Signal catching | Enabled | c |
| Concurrent garbage collection mark | Disabled | c |
| Garbage collection heap compaction | Enabled | c |
| Number of garbage collection helper threads | (Number of processors – 1) | c |
| Initial size of system heap | 128 KB on 32-bit<br>8 MB on 64-bit | c |
| Maximum heap expansion size/ratio | Zero | c |
| Maximum heap free space ratio | 60% | c |
| Minimum heap expansion size/ratio | 1 MB | c |

*Table 34. Cross platform defaults (continued)*

| JVM setting | Default | Setting affected by |
|---|---|---|
| Minimum heap free space ratio | 30% | c |
| Garbage collection of classes | Enabled | c |
| Classpath | Not set | ec |
| Java options cache | Not used | e |
| Accessibility support | Enabled | e |
| JIT | Enabled | ec |
| JIT debug options | Disabled | e |
| MMI | Enabled | e |
| 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 |

*Table 35. Platform specific defaults*

| JVM Setting | AIX | Linux | Windows | z/OS | Setting affected by |
|---|---|---|---|---|---|
| Minimum heap size | 4 MB | 4 MB | 4 MB | 1 MB | c |
| Maximum heap size | 64 MB | Note 1 | Note 2 | 64 MB | c |
| Native stack size for any thread | 512 KB | 256 KB | 1 MB | 524 KB | c |
| Default locale | None | None | N/A | None | e |
| Path for code library load | None | None | N/A | None | e |
| Time to wait before starting plug-in | N/A | Zero | N/A | N/A | e |
| Debug malloc trace | Off | Off | N/A | Off | e |
| Temporary directory | /tmp | /tmp | \tmp | /tmp | e |
| Size of application class system heap | N/A | N/A | N/A | Note 3 | c |
| Initial transient heap size | N/A | N/A | N/A | Note 4 | c |
| Master JVM initial size | N/A | N/A | N/A | 1 MB | c |
| 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 |
| Transaction dumps | N/A | N/A | N/A | Enabled | e |
| Max transaction dumps | N/A | N/A | N/A | 2 | e |
| Transaction dump data/set name | N/A | N/A | N/A | Note 5 | e |
| Inherit thread address space | N/A | N/A | N/A | Enabled | e |
| Thread model | N/A | N/A | N/A | Native | e |
| Redirect JVM stderr output | N/A | N/A | N/A | Disabled | e |
| Assume kernel | N/A | Not set | N/A | N/A | e |

**Notes:**

1. The heap grows to half of physical memory, but always to a minimum of 16 MB and not beyond (512 MB – 1).
2. The heap grows to half of physical memory, but always to a minimum of 16 MB and not beyond (2 GB – 1).
3. Resettable JVM only. 128 KB on 32-bit, 8 MB on 64-bit.
4. Resettable JVM only. Half of minimum heap if specified: otherwise, 512 KB.
5. Default setting is %s.JVM.TDUMP.&JOBNAME..D&YYMMDD..T& HHMMSS.

# Appendix I. Using the alternative JVM for Java debugging

The IBM SDK contains a distinct alternative debug environment that is provided, for some platforms, to be used *only* for Java application debugging. This debug environment is provided on:

- AIX PPC32 and PPC64
- Linux PPC32 and PPC64

It is also provided on Linux IA32 and Windows IA32 when they are shipped as products based on IBM Rational Application Developer for WebSphere, and is used as the debugging environment for WebSphere Application Server.

The debug environment is very similar to the default environment in the way that it uses the Java class libraries. It differs, however, in its JVM and JIT characteristics, to the extent that many of the options and facilities that are described in this *Diagnostics Guide* are not directly applicable. In the debug environment:

- A different garbage collection algorithm is used.
- The dump format is different.
- Some command-line options are different.
- Some command-line options have a different effect.

A few classes (such as Object, Class, and Thread) are intrinsically linked to the implementation of the runtime environment. When the Java launcher is invoked with the alternative debug configuration, a replacement set of these classes is prepended to the BOOTCLASSPATH. These alternatives override the default implementations.

The alternative environment is enabled when:

- The **-Xdebug** command-line option is specified (AIX and Linux PPC32 and PPC64 architectures only).
- The **-Xj9** command-line option is specified (Linux IA32 and Windows IA32 only, when they are shipped with IBM Rational Application Developer for WebSphere).

To determine which environment you are using and its build date, use the **java -version** command. Here is an example of the output that you can expect from the default runtime environment on Windows IA32:

```
C:\>java -version
java version "1.4.2"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2)
Classic VM (build 1.4.2, J2RE 1.4.2 IBM Windows 32 build cn142-20060330 (JIT enabled: jitc))
```

Here is an example of the output you can expect from the alternative debug environment on Windows IA32:

```
C:\>java -Xj9 -version
java version "1.4.2"
Java(TM) 2 Runtime Environment, Standard Edition (build 2.2)
IBM J9SE VM (build 2.2, J2RE 1.4.2 IBM J9 2.2 Windows 2000 x86-32 j9n142-20060330 (JIT enabled)
J9VM - 20060325_1559_lHdSMR
JIT  - r7_level20060321_1801)
```

Here is an example of the output you can expect from the alternative debug environment on AIX PPC64:

```
$ java -Xdebug -version
java version "1.4.2"
Java(TM) 2 Runtime Environment, Standard Edition (build 2.3)
IBM J9 VM (build 2.3, J2RE 1.4.2 IBM J9 2.3 AIX ppc64-64 j9ap64142-20060329 (JIT enabled)
J9VM - 20060327_05972_BHdSMr
JIT  - 20060323_1800_r8
GC   - 20060323_AA)
```

**Note:** The **java -fullversion** command always returns the version information for the default runtime environment. You must use the **-version** flag to return the debug environment. You can get this information at runtime by using System.getProperties() or by using the system properties that are shown in Table 36. In the table, the values for the system properties are examples only.

*Table 36. System properties*

| Command | Default | Alternative |
|---|---|---|
| java.vm.name | Classic VM | IBM J9SE VM |
| java.fullversion | J2RE 1.4.1 IBM Windows 32 build cn141–20030711 (JIT enabled:jitc) | J2RE 1.4.1 IBM J9 build 20030705 (JIT enabled) |

The debug environment runs with most of the optimizations provided by the JIT. To disable the JIT, use the **-Xint** option

# How the debug environment relates to other components

Serviceability features in the debug environment differ from the ones in the default runtime environment.

## Dumps

The debug environment supports the production of diagnostics files using the **-Xdump** option. For more information, use **-Xdump:help**. For detailed information about using dumps on the Intel debug platforms, see the *Diagnostics Guide, 1.4.2, for z/OS64 and AMD™64 platforms*, and, for the PPC platforms, see the *Diagnostics Guide, 5.0* at http://www.ibm.com/developerworks/java/jdk/diagnosis.

To analyze core dumps produced in the debug environment, use the j9jextract tool instead of the jextract tool to process the dump and the jdmpview tool instead of the jformat tool.

## Trace

The debug environment supports the production of trace output using the **-Xtrace** option. However, tracing of class library natives and application trace are not supported. For detailed information about using trace on the Intel debug platforms, see the *Diagnostics Guide, 1.4.2, for z/OS64 and AMD™64 platforms*, and, for the PPC platforms, see the *Diagnostics Guide, 5.0* at http://www.ibm.com/developerworks/java/jdk/diagnosis.

Any trace files produced using the debug environment must also be formatted using the debug environment. For example, if you generate a trace file using the debug environment, like this: `java -Xj9 -Xtrace:maximal=all,output=debugEnvTraceFile HelloWorld`, you must format it

using the debug environment as follows: `java -Xj9 com.ibm.jvm.format.TraceFormat debugEnvTraceFile`. If you try to format a debug environment generated trace file using the non-debug environment, the JVM will cause runtime errors and the file will not be formatted.

### Verbose garbage collection

In the debug environment, the **-verbose:gc** option produces garbage collection information. The format of this verbose GC output is different from the output produced in the default environment. For detailed information about verbose garbage collection on the Intel debug platforms, see the *Diagnostics Guide, 1.4.2, for z/OS64 and AMD™64 platforms*, and, for the PPC platforms, see the *Diagnostics Guide, 5.0* at http://www.ibm.com/developerworks/java/jdk/diagnosis.

### JNIChk utility

The debug environment provides the **-Xcheck:jni**option to add sanity checking to the Java Native Interface (JNI). Performance is affected because every JNI call does extra checks on the parameters. This check can, however, be very useful when you suspect that the JNI code that is associated with the framework or application might have bugs. For detailed information about the JNIChk utility on the Intel debug platforms, see the *Diagnostics Guide, 1.4.2, for z/OS64 and AMD™64 platforms*, and, for the PPC platforms, see the *Diagnostics Guide, 5.0* at http://www.ibm.com/developerworks/java/jdk/diagnosis.

### The JIT

In the debug environment, you can disable the JIT by using the **-Xint** command-line option. If a problem no longer occurs after you have used the **-Xint** command-line option to disable the JIT, the problem is probably related to the JIT or to one of its optimizations. For instructions about how to isolate JIT problems on the Intel debug platforms, see the *Diagnostics Guide, 1.4.2, for z/OS64 and AMD™64 platforms*, and, for the PPC platforms, see the *Diagnostics Guide, 5.0* at http://www.ibm.com/developerworks/java/jdk/diagnosis.

## Command-line options in the debug environment

Appendix G, "Command-line parameters," on page 487 describes the command-line and system properties supported by the default runtime environment. In general, the same set of options and properties are supported by the alternative debug environment. Differences are described in Table 37.

For the full description of options available in the debug environment, on the Intel debug platforms, see the *Diagnostics Guide, 1.4.2, for z/OS64 and AMD™64 platforms*, and, for the PPC platforms, see the *Diagnostics Guide, 5.0* at http://www.ibm.com/developerworks/java/jdk/diagnosis.

*Table 37. Command-line differences*

| Command line arguments | Debug environment behavior |
|---|---|
| -Dibm.ci.verbose | Ignored. |
| -Dibm.cl.eagerresolution | Ignored. |
| -Dibm.dg.trc.* | Ignored. Use -Xtrace options. |
| -Djava.compiler={ NONE \| jitc } | Deprecated. Use -Xint and -Xjit. |
| -verbose:Xclassdep | Not supported. JVM will terminate with error message. |

*Table 37. Command-line differences (continued)*

| Command line arguments | Debug environment behavior |
|---|---|
| -version | Prints version information for the alternative runtime environment. |
| -showversion | Prints version information for the alternative runtime environment. |
| -Xcheck:nabounds | Equivalent to -Xchecxk:jni. |
| -Xinitsh | Ignored. |
| -Xoss | Deprecated. Use -Xmso. |
| -Xpd | Not supported. JVM will terminate with error message. |
| -Xtlhs | Not supported. JVM will terminate with error message. |

# Appendix J. Using a Problem Determination build of the JVM

The Problem Determination (PD) build of the JVM comprises a set of libraries built with additional PD function to help IBM support deal with some categories of customer problems. These libraries are fully compatible with the Java launchers and with customer launchers built with the JNI invocation function. The PD build has passed Java Compatibility testing and has been system tested, and is therefore of comparable quality to the normal build of the JVM.

The PD build is a build of the native libraries in the JVM. When it is used, the normal Java classes are executed from their normal locations.

## When to use the PD build

The PD build is intended to be used under the guidance of IBM support to assist in resolving problems that seem to relate to the native libraries in the core JVM and the supporting libraries containing native methods for the standard Java classes (for example, the java/net package). The PD build is not a customer-orientated tool and should be used only when an IBM support team requests it.

## Why is the PD build necessary?

For performance reasons, only a limited amount of tracing and sanity-checking code is included in the normal build of the IBM virtual machine for Java. Some situations can be more quickly resolved by a build with more PD function. Previously, the "_g" build provided this function, but it was difficult to use in many situations because both launchers and libraries had nonstandard names ("java_g", "libjvm_g.a", ...). These libraries were also built with different compiler optimization levels to the normal build, which affected performance and often changed timing enough to "chase away" problems under investigation.

The PD build addresses these limitations by using the normal launchers and the same library names as the normal build, and the libraries are compiled with the normal level of optimization.

## Where to find the PD build

The PD build will be supplied by IBM support when required.

The PD build libraries have the same names as the normal JVM libraries, but are installed into a different directory subtree:

- The normal libraries reside in the bin subdirectory of your JRE installation and the classic subdirectory of bin. So, for example, on AIX you would find .../jre/bin/classic/libjvm.a.
- The PD build libraries add another layer of subdirectory at the top of the tree, called pd. So, on AIX again, you would find .../jre/pd/bin/classic/libjvm.a.

## How to enable the PD build

To run a standard Java launcher command with the PD build, add the **-Xpd** option to the command-line arguments for the launcher. For example **java -Xpd** ***class*** or **jformat -J-Xpd** ***sdff_file***. The **-Xpd** option must be specified before any other parameter; if it is not placed first, the JVM might report an `unrecognized option error`.

If you have a custom launcher using the JNI invocation interface, you must ensure that your launcher loads the JVM libraries from the pd subdirectories of the JRE installation, normally by adjusting the environment variable controlling how the operating system finds shared libraries for loading. On AIX and z/OS, you would prepend the .../jre/pd/bin and .../jre/pd/bin/classic directories to your LIBPATH, for example, while on Linux you would change your LD_LIBRARY_PATH

# Appendix K. Some notes on jformat and the jvmdcf file

**This appendix is likely to be of interest to you only if you are involved in advanced, low-level debugging activities. The information is subject to change without notice, and there is no guarantee it will be maintained or provided in future.**

To allow the jformat utility to understand the structures (control blocks and the fields within them) in specific builds, file jvmdcf in the jre/bin directory provides layout information. This file is a zip file containing jvmdcf.X, which is a file obtained from the debug compile/link steps of a simple file that contains `#include` and various typedef information. The jformat tool uses jvmdcf.X to understand the layout and types of fields within control blocks. jvmdcf is loaded into memory in a compressed form when the jvm is established and thus is present in the dump. You find it by scanning memory looking for the ASCII characters JVMRAS starting on an 8-byte boundary, followed by X'000055AA55AA55AA55AA' on little-endian platforms and X'0000AA55AA55AA55AA55' on big-endian platforms. The examples here are from a little-endian system. The JVMRAS is code in the extractors that finds and stores away the address in memory and size of the jvmdcf image.

The internal structure of jvmdcf.X is logically equivalent to an 8-byte character field containing the value JVMSYMS, flowed by unsigned integer fields for version, encoding, number of symbols, symbol offset, symbol size, string offset, and string size.

The value JVMSYMS can be seen from this display showing the start of a jvmdcf.X file:

| Offset | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00000000 | 4A | 56 | 4D | 53 | 59 | 4D | 53 | 00 | 00 | 01 | 00 | 01 | 00 | 00 | 00 | 00 | JVMSYMS..... |
| 00000010 | 00 | 00 | 06 | CE | 00 | 00 | 00 | 24 | 00 | 00 | 88 | 18 | 00 | 00 | 88 | 3C | ...Î...$...ˆ...ˆ< |
| 00000020 | 00 | 01 | 93 | FD | 00 | 00 | 06 | CD | 00 | 00 | 00 | 04 | 00 | 00 | 00 | 00 | ..“ý...Í..... |
| 00000030 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 18 | 00 | 00 | 01 | FA | 00 | 00 | 00 | 02 | ..........ú.... |
| 00000040 | 00 | 00 | 00 | 30 | 00 | 00 | 04 | CD | 00 | 00 | 04 | D7 | 00 | 00 | 01 | FB | ...0...Í...×...û |
| 00000050 | 00 | 00 | 00 | 02 | 00 | 00 | 00 | 30 | 00 | 00 | 05 | AF | 00 | 00 | 05 | B9 | .......0...¯...¹ |
| 00000060 | 00 | 00 | 01 | FC | 00 | 00 | 00 | 02 | 00 | 00 | 00 | 30 | 00 | 00 | 05 | E1 | ...ü.......0...á |
| 00000070 | 00 | 00 | 05 | EB | 00 | 00 | 01 | FF | 00 | 00 | 00 | 02 | 00 | 00 | 00 | 30 | ...ë...ÿ.......0 |
| 00000080 | 00 | 00 | 05 | FE | 00 | 00 | 06 | 08 | 00 | 00 | 02 | 00 | 00 | 00 | 00 | 02 | ...þ............ |
| 00000090 | 00 | 00 | 00 | 34 | 00 | 00 | 06 | 1B | 00 | 00 | 06 | 25 | 00 | 00 | 02 | 01 | ...4.......%.... |
| 000000A0 | 00 | 00 | 00 | 02 | 00 | 00 | 00 | 38 | 00 | 00 | 07 | 07 | 00 | 00 | 07 | 11 | .......8........ |
| 000000B0 | 00 | 00 | 02 | 02 | 00 | 00 | 00 | 02 | 00 | 00 | 00 | 38 | 00 | 00 | 07 | 39 | ............8...9 |

*Figure 15. The start of a jvmdcf.X file*

In the example above, the number of symbols is `0x000006CE` (= 1742) and the symbol offset is x24. Starting at the end of the symbol header is an array of symbol table entries, where each entry is based on a structure of integers for type number, base type, size, name offset, and description offset.

Base types as referenced in this appendix are:

* pointer (5)

**s** structure (2)

**u** union (3)

**e** enum (4)

**a** array (6)

**f** function (7)

**-** primitives ( int, char, long, unsigned long...) (1)

Numbers refer to the value in the individual symbol table entries and the letters are used in the null-terminated strings area.

Here, 1742*20+36 = 0x883C (which is the string offset field within the symbol header structure). The string offset points to an array of null-terminated strings, the structure of which is described below.

| Offset | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00008830 | 00 | 00 | 00 | 04 | 00 | 01 | 93 | F8 | 00 | 01 | 93 | FC | 5F | 4A | 56 | 4D | ......▌ø..▌ü_JVM |
| 00008840 | 43 | 6F | 6E | 73 | 74 | 61 | 6E | 74 | 44 | 65 | 66 | 69 | 6E | 69 | 74 | 69 | ConstantDefiniti |
| 00008850 | 6F | 6E | 73 | 00 | 65 | 2D | 31 | 43 | 4F | 4E | 53 | 54 | 41 | 4E | 54 | 5F | ons.e-1CONSTANT_ |
| 00008860 | 55 | 74 | 66 | 38 | 3A | 31 | 2C | 43 | 4F | 4E | 53 | 54 | 41 | 4E | 54 | 5F | Utf8:1,CONSTANT_ |
| 00008870 | 55 | 6E | 69 | 63 | 6F | 64 | 65 | 3A | 32 | 2C | 43 | 4F | 4E | 53 | 54 | 41 | Unicode:2,CONSTA |
| 00008880 | 4E | 54 | 5F | 49 | 6E | 74 | 65 | 67 | 65 | 72 | 3A | 33 | 2C | 43 | 4F | 4E | NT_Integer:3,CON |
| 00008890 | 53 | 54 | 41 | 4E | 54 | 5F | 46 | 6C | 6F | 61 | 74 | 3A | 34 | 2C | 43 | 4F | STANT_Float:4,CO |
| 000088A0 | 4E | 53 | 54 | 41 | 4E | 54 | 5F | 4C | 6F | 6E | 67 | 3A | 35 | 2C | 43 | 4F | NSTANT_Long:5,CO |
| 000088B0 | 4E | 53 | 54 | 41 | 4E | 54 | 5F | 44 | 6F | 75 | 62 | 6C | 65 | 3A | 36 | 2C | NSTANT_Double:6, |
| 000088C0 | 43 | 4F | 4E | 53 | 54 | 41 | 4E | 54 | 5F | 43 | 6C | 61 | 73 | 73 | 3A | 37 | CONSTANT_Class:7 |
| 000088D0 | 2C | 43 | 4F | 4E | 53 | 54 | 41 | 4E | 54 | 5F | 53 | 74 | 72 | 69 | 6E | 67 | ,CONSTANT_String |
| 000088E0 | 3A | 38 | 2C | 43 | 4F | 4E | 53 | 54 | 41 | 4E | 54 | 5F | 46 | 69 | 65 | 6C | :8,CONSTANT_Fiel |
| 000088F0 | 64 | 72 | 65 | 66 | 3A | 39 | 2C | 43 | 4F | 4E | 53 | 54 | 41 | 4E | 54 | 5F | dref:9,CONSTANT_ |
| 00008900 | 4D | 65 | 74 | 68 | 6F | 64 | 72 | 65 | 66 | 3A | 31 | 30 | 2C | 43 | 4F | 4E | Methodref:10,CON |
| 00008910 | 53 | 54 | 41 | 4E | 54 | 5F | 49 | 6E | 74 | 65 | 72 | 66 | 61 | 63 | 65 | 4D | STANT_InterfaceM |
| 00008920 | 65 | 74 | 68 | 6F | 64 | 72 | 65 | 66 | 3A | 31 | 31 | 2C | 43 | 4F | 4E | 53 | ethodref:11,CONS |
| 00008930 | 54 | 41 | 4E | 54 | 5F | 4E | 61 | 6D | 65 | 41 | 6E | 64 | 54 | 79 | 70 | 65 | TANT_NameAndType |
| 00008940 | 3A | 31 | 32 | 2C | 41 | 43 | 43 | 5F | 50 | 55 | 42 | 4C | 49 | 43 | 3A | 31 | :12,ACC_PUBLIC:1 |
| 00008950 | 2C | 41 | 43 | 43 | 5F | 50 | 52 | 49 | 56 | 41 | 54 | 45 | 3A | 32 | 2C | 41 | ,ACC_PRIVATE:2,A |
| 00008960 | 43 | 43 | 5F | 50 | 52 | 4F | 54 | 45 | 43 | 54 | 45 | 44 | 3A | 34 | 2C | 41 | CC_PROTECTED:4,A |
| 00008970 | 43 | 43 | 5F | 53 | 54 | 41 | 54 | 49 | 43 | 3A | 38 | 2C | 41 | 43 | 43 | 5F | CC_STATIC:8,ACC_ |
| 00008980 | 46 | 49 | 4E | 41 | 4C | 3A | 31 | 36 | 2C | 41 | 43 | 43 | 5F | 53 | 59 | 4E | FINAL:16,ACC_SYN |
| 00008990 | 43 | 48 | 52 | 4F | 4E | 49 | 5A | 45 | 44 | 3A | 33 | 32 | 2C | 41 | 43 | 43 | CHRONIZED:32,ACC |
| 000089A0 | 5F | 53 | 55 | 50 | 45 | 52 | 3A | 33 | 32 | 2C | 41 | 43 | 43 | 5F | 56 | 4F | _SUPER:32,ACC_VO |
| 000089B0 | 4C | 41 | 54 | 49 | 4C | 45 | 3A | 36 | 34 | 2C | 41 | 43 | 43 | 5F | 54 | 52 | LATILE:64,ACC_TR |
| 000089C0 | 41 | 4E | 53 | 49 | 45 | 4E | 54 | 3A | 31 | 32 | 38 | 2C | 41 | 43 | 43 | 5F | ANSIENT:128,ACC_ |
| 000089D0 | 4E | 41 | 54 | 49 | 56 | 45 | 3A | 32 | 35 | 36 | 2C | 41 | 43 | 43 | 5F | 49 | NATIVE:256,ACC_I |
| 000089E0 | 4E | 54 | 45 | 52 | 46 | 41 | 43 | 45 | 3A | 35 | 31 | 32 | 2C | 41 | 43 | 43 | NTERFACE:512,ACC |
| 000089F0 | 5F | 41 | 42 | 53 | 54 | 52 | 41 | 43 | 54 | 3A | 31 | 30 | 32 | 34 | 2C | 41 | _ABSTRACT:1024,A |
| 00008A00 | 43 | 43 | 5F | 53 | 54 | 52 | 49 | 43 | 54 | 3A | 32 | 30 | 34 | 38 | 2C | 41 | CC_STRICT:2048,A |
| 00008A10 | 43 | 43 | 5F | 56 | 41 | 4C | 4B | 4E | 4F | 57 | 4E | 3A | 38 | 31 | 39 | 32 | CC_VALKNOWN:8192 |
| 00008A20 | 2C | 41 | 43 | 43 | 5F | 4D | 41 | 43 | 48 | 49 | 4E | 45 | 5F | 43 | 4F | 4D | ,ACC_MACHINE_COM |

*Figure 16. A symbol table entry*

Examining the first symbol table entry above:

- The type number is `0x6CD - 1741` (this is a reference count so that structures can be linked together).
- The base type is 4 (meaning enum).

- The size is 0.
- The name offset is 0 (this is an offset from the start of the strings pointed to by string offset).
- The description offset is 0x18 (=24) from the string offset.

Displaying position 0x883c in the jvmdcf.X file, you can see that this position points to a structure definition (of zero size because it is recording some constant values) for **_JVMConstantDefinitions**. Next is a manufactured string (at offset 0x18 from the start of the strings) representing the fields (constant values) within there (the e-1 indicates that an enum values list follows). Then there is a large string (terminated with 0x00 at 0x8d08 into this particular jvmdcf.X) which is essentially a list of constants and their values. The description is given with the definition of value for the enum list, thus:

```
e-1CONSTANT_Utf8:1,CONSTANT_Unicode:2..
```

as exemplified in Figure 16 on page 506 means that the symbol **CONSTANT_Utf8** is an integer 1, **CONSTANT_Unicode** is 2, and so on.

It is more interesting to find the entry for a "well-known" control block in the JVM and look for that. One such is "Jvm". You can use the **jformat** command **dis CB** to find the control blocks known to a dump which relies on the information within the memory embedded jvmdcf). The Jvm control blocks expand to a complex structure – the commands **jformat for jvm** or **dis cbo(Jvm)**, described in "Using jformat to display the JVM control block" on page 508, demonstrate this.

The null-terminated string "Jvm" is at offset 0x1cf40 into this particular jvmdcf.X, so it should have a name offset in an symbol table entry of 0x14704 (0x1cf40-0x883C), and there is a symbol table entry that matches this at 0x6690 into jvmdcf.X, which looks like:

```
00006690 | 00 00 01 2A 00 00 00 02  00 00 0A D4 00 01 47 04
000066A0 | 00 01 47 08 00 00 07 8A  00 00 00 05 00 00 00 04
```

You can see from this symbol table entry that:

- The type number is 0x12A.
- The base type is 2 (meaning structure).
- The length is 0xad4 (2772).
- The name offset is 0x14704 (from 0x883c).
- The description offset is 0x14708.

Now consider what the base type in the symbol table entry shows.

The file from offset 0x1cf40 onwards is displayed below.

| Offset | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0001CF40 | 4A | 76 | 6D | 00 | 73 | 32 | 37 | 37 | 32 | 68 | 65 | 61 | 64 | 65 | 72 | 3A | Jvm.s2772header: |
| 0001CF50 | 33 | 33 | 32 | 2C | 30 | 2C | 31 | 32 | 38 | 3B | 76 | 6D | 3A | 31 | 36 | 39 | 332,0,128;vm:169 |
| 0001CF60 | 35 | 2C | 31 | 32 | 38 | 2C | 33 | 32 | 3B | 66 | 61 | 63 | 61 | 64 | 65 | 3A | 5,128,32;facade: |
| 0001CF70 | 33 | 34 | 33 | 2C | 31 | 36 | 30 | 2C | 32 | 31 | 38 | 38 | 38 | 3B | 6A | 61 | 343,160,21888;ja |
| 0001CF80 | 62 | 3A | 34 | 32 | 38 | 30 | 2C | 32 | 32 | 30 | 34 | 38 | 2C | 33 | 32 | 3B | b:4280,22048,32; |
| 0001CF90 | 6D | 6F | 64 | 65 | 3A | 39 | 2C | 32 | 32 | 30 | 38 | 30 | 2C | 33 | 32 | 3B | mode:9,22080,32; |
| 0001CFA0 | 66 | 75 | 6C | 6C | 56 | 65 | 72 | 73 | 69 | 6F | 6E | 3A | 35 | 36 | 2C | 32 | fullVersion:56,2 |
| 0001CFB0 | 32 | 31 | 31 | 32 | 2C | 33 | 32 | 3B | 73 | 75 | 66 | 66 | 69 | 78 | 3A | 35 | 2112,32;suffix:5 |
| 0001CFC0 | 36 | 2C | 32 | 32 | 31 | 34 | 34 | 2C | 33 | 32 | 3B | 3B | 00 | 28 | 6E | 75 | 6,22144,32;;.(nu |

*Figure 17. The file from offset 0x1cf40*

Figure 17 shows starts with `S2772`, indicating that it is a structure of length 2772. Next, the first major stanza in the description string is `header:332,0,128`, which identifies the first field within the JVM control block named "header", and is defined further using the symbol table entry with type number 332 (`0x14C`). You can find this symbol table entry at position `0x667C` into the file and it points to information at `0x0001CEE4`, which contains the structure details for a field called JvmDataHeader. And so the chain continues until you reach fields that are not structures (or unions).

The 0 and 128 in the field entry for the header field are intended to be offsets and length, but these are not currently reliable.

## Using jformat to display the JVM control block

For details about jformat, see "Analyzing dumps with jformat" on page 263.

The information given here should enable you to understand the format of the jvmdcf.X file. This file is effectively read in jformat from dump memory. (Note that it is compressed in memory.) When the jformat **dis cb** and **dis cbo(cbname)** commands are used to look at control blocks, the deconstructed information from jvmdcf.X is used to produce the output. Similarly, jformat with **for** *address* **as** *controlblock* causes stored objects built at dump open time to be used for display of the specified control block. For example, the command **for 0x10144a40 as Jvm** on Mywin1.sdff might give the following output (shortened for convenience as the JVM is a large control block).

```
for 0x10144a40 as Jvm

 ......command executing
Jvm @ 0x10144a40
  header :    @ 0x10144a40
    eyecatcher @ 0x10144a40 (array)
    length = 0xae4
    version = 0x1
    modification = 0x0
  vm = 0x1010aa7c
  facade :    @ 0x10144a54
    lk :      @ 0x10144a54
      header :        @ 0x10144a54
        eyecatcher @ 0x10144a54 (array)

 ...
 ...
 ...

  jab = 0x235568
  mode = 0x0
```

```
  fullVersion = 0x1010a4a4
  suffix = 0x1012e690

 as Jvm finished
Ready......
```

The version level held within memory will be a C-style string (null terminated) and, depending on the build level, will be `J2RE 1.4.2 IBM Windows 32 build cn142-20040608` as returned by the command **java -version**:

```
java version "1.4.2"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2)
Classic VM (build 1.4.2, J2RE 1.4.2 IBM Windows 32 build cn142-20040608
(JIT enabled: jitc))
```

# Appendix L. 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.

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 | AS/400 |
| CICS | DB/2 |
| IBM | OS/2 |
| OS/2 Warp | OS/390 |
| WebSphere | z/OS |

## Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product and service names may be trademarks or service marks of others.

# Index

## Special characters



## Numerics



## A

# B

# C

# D

# E



# F

# G

# H

# I



# J

# K



# L

# M

# N



# O

# P

# Q



# R

# S

# T

# U

# V



# W



# X

# Z

SC34-6358-06