# Java Platform, Standard Edition Troubleshooting Guide



Release 11 E94880-08 April 2024

ORACLE

Java Platform, Standard Edition Troubleshooting Guide, Release 11

E94880-08

Copyright © 1995, 2024, Oracle and/or its affiliates.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at <a href="http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc">http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc</a>.

# Contents

#### Preface

Audience	xi
Documentation Accessibility	xi
Related Documents	xi
Conventions	xi

# Part I General Java Troubleshooting

## 1 Prepare Java for Troubleshooting

Set Up Java for Troubleshooting	1-1
Enable Options and Flags for JVM Troubleshooting	1-1
Gather Relevant Data	1-3
Make a Java Application Easier to Debug	1-3

## 2 Diagnostic Tools

Diagnostic Tools Overview	2-2
JDK Mission Control	2-2
Troubleshoot with JDK Mission Control	2-3
Flight Recorder	2-4
Produce a Flight Recording	2-5
Start a Flight Recording	2-5
Use Triggers for Automatic Flight Recordings	2-6
Use Startup Flags at the Command Line to Produce a Flight Recording	2-7
Analyze a Flight Recording	2-8
Analyze a Flight Recording Using JMC	2-8
Analyze a Flight Recording Using the jfr tool or JFR APIs	2-11
The jcmd Utility	2-12
Useful Commands for the jcmd Utility	2-14
Troubleshoot with the jcmd Utility	2-15
Native Memory Tracking	2-16



Use NMT to Detect a Memory Leak	2-16
How to Monitor VM Internal Memory	2-16
NMT Memory Categories	2-24
JConsole	2-25
Troubleshoot with the JConsole Tool	2-25
Monitor Local and Remote Applications with JConsole	2-27
The jdb Utility	2-28
Troubleshoot with the jdb Utility	2-29
The jinfo Utility	2-29
Troubleshooting with the jinfo Utility	2-31
The jmap Utility	2-31
Heap Configuration and Usage	2-32
Heap Histogram	2-33
Class Loader Statistics	2-35
The jps Utility	2-36
The jstack Utility	2-37
Troubleshoot with the jstack Utility	2-37
Stack Trace from a Core Dump	2-37
Mixed Stack	2-38
The jstat Utility	2-40
The visualgc Tool	2-42
Control+Break Handler	2-43
Thread Dump	2-43
Thread States for a Thread Dump	2-45
Detect Deadlocks	2-45
Heap Summary	2-46
Native Operating System Tools	2-46
DTrace Tool	2-47
Probe Providers in Java HotSpot VM	2-47
Improvements to the pmap Utility	2-48
Improvements to the pstack Utility	2-49
Custom Diagnostic Tools	2-49
Java Platform Debugger Architecture	2-49
Postmortem Diagnostic Tools	2-50
Hung Processes Tools	2-51
Monitoring Tools	2-52
Other Tools, Options, Variables, and Properties	2-53
The java.lang.management Package	2-54
The java.lang.instrument Package	2-54
The java.lang.Thread Class	2-54
JVM Tool Interface	2-55



The jrunscript Utility	2-55
The jstatd Daemon	2-55
Troubleshooting Tools Based on the Operating System	2-55

## 3 Troubleshoot Memory Leaks

3-1
3-1
3-3
3-4
3-7
3-10
3-10
3-11
3-12
3-13
3-13
3-14
3-15
3-15
3-17

## 4 Troubleshoot Performance Issues Using Flight Recorder

Flight Recorder Overhead	4-1
Use JDK Mission Control to Find Bottlenecks	4-2
Use JDK Mission Control to Debug Garbage Collection Issues	4-3
Use JDK Mission Control to Debug Synchronization Issues	4-5
Use JDK Mission Control to Debug I/O Issues	4-6
Use JDK Mission Control to Monitor Code Execution Performance	4-7

# Part II Debug JVM Issues

## 5 Troubleshoot System Crashes

Determine Where the Crash Occurred	5-1
Crash the Native Code	5-1
Crash in the Compiled Code	5-3
Crash in the HotSpot Compiler Thread	5-4
Crash in the VM Thread	5-4
Crash Due to Stack Overflow	5-4



Find a Workaround	5-6
Working Around Crashes in the HotSpot Compiler Thread or Compiled Code	5-6
Working Around Crashes During Garbage Collection	5-8
Working Around Crashes Caused by Class Data Sharing	5-10
Microsoft Visual C++ Version Considerations	5-10

## 6 Troubleshoot Process Hangs and Loops

Diagnose a Loop Process	6-1
Diagnose a Hung Process	6-2
Deadlock Detected	6-2
Deadlock Not Detected	6-4
No Thread Dump	6-4
Oracle Solaris 8 Thread Library	6-6

## 7 Handle Signals and Exceptions

Handle Signals on Oracle Solaris, Linux, and macOS	7-1
Handle Exceptions on Windows	7-1
Signal Chaining	7-3
Handle Exceptions Using the Java HotSpot VM	7-5
Console Handlers	7-5
Signals Used in Oracle Solaris, Linux, and macOS	7-6

# Part III Debug Core Library Issues

8	Time Zone	Settings	in the	JRE
---	-----------	----------	--------	-----

Native Time Zone Information and the JRE	
Determine the Time Zone Data Version in Use	8-1
Troubleshoot Problems with TZupdater	8-2
Determine the Default Time Zone on Windows	8-3
Check the Default Time Zone Java Runtime Reports	8-3
Determine the Setting in the Control Panel	8-3
Check for Automatic Daylight Saving Time Adjustment	8-4
Set the Default Time Zone in Windows Settings	8-5
Check -Duser.timezone System Property	8-5
Special Tool in Windows	8-5
Internal Representation of Time Zone Mappings	8-6



# Part IV Debug Client Issues

# 9 Introduction to Client Issues

Java SE Desktop Technologies	
General Steps to Troubleshoot an Issue	
Identify the Type of Issue	9-3
Java Client Crashes	9-4
Performance Problems	9-4
Behavior Problems	9-5
Basic Tools	9-6
Java Debug Wire Protocol	

### 10 AWT

Debug Tips for AWT	10-1
Layout Manager Issues	10-2
Key Events	10-2
Modality Issues	10-3
AWT Crashes	10-3
Focus Events	10-5
How to Trace Focus Events	10-5
Native Focus System	10-6
Focus Models Supported by X Window Managers	10-7
Miscellaneous Problems with Focus	10-7
Data Transfer	10-9
Debug Drag-and-Drop Applications	10-9
Frequent Issues with Data Transfer	10-9
Other Issues	10-11
Splash Screen Issues	10-12
Tray Icon Issues	10-12
Pop-up Menu Issues	10-12
Background or Foreground Color Inheritance	10-13
AWT Panel Size Restriction	10-13
Hangs During Debugging of Pop-up Menus and Similar Components on X11	10-13
Window.toFront()/toBack() Behavior on X11	10-14
Heavyweight or Lightweight Components Mix	10-14



# 11 Java 2D Pipeline Rendering and Properties

Oracle Solaris and Linux: X11 Pipeline	11-1
X11 Pipeline Pixmaps Properties	11-2
X11 Pipeline MIT Shared Memory Extension	11-3
Oracle Solaris on SPARC: DGA Support	11-3
Oracle Solaris on SPARC - Change Java 2D Default Visual	11-4
Windows OS: DirectDraw/GDI Pipeline	11-4
Windows OS: Direct3D Pipeline in Full-Screen Mode	11-6
OpenGL Pipeline in Oracle Solaris, Linux, and Windows	11-7
Enable OpenGL Pipeline	11-7
Minimum Requirements	11-7
Diagnose Startup Issues	11-8
Diagnose Rendering and Performance Issues	11-9
Latest OpenGL Drivers	11-9

## 12 Java 2D

Generic Performance Issues	12-1
Hardware-Accelerated Rendering Primitives	12-1
Primitive Tracing to Detect and Avoid Non-Accelerated Rendering	12-2
Causes of Poor Rendering Performance	12-3
Improve Performance of Software-only Rendering	12-5
Text-Related Issues	
Application Crash During Text Rendering	12-6
Differences in Text Appearance	12-8
Metrics	12-9
Java 2D Printing	12-9

## 13 Swing

General Debug Tips for Swing	13-1
Specific Debug Tips for Swing	13-2
Incorrect Threading	13-2
JComponent Children Overlap	13-3
Display Update	13-4
Model Change	13-4
Add or Remove Components	13-4
Opaque Override	13-4
Permanent Changes to Graphics	13-4
Custom Painting and Double Buffering	13-5
Opaque Content Pane	13-5



Renderer Call for Each Cell Performance	13-5
Possible Leaks	13-5
Mix Heavyweight and Lightweight Components	13-6
Use Synth	13-6
Track Activity on Event Dispatch Thread	13-6
Specify Default Layout Manager	13-6
Listener Object Dispatched to Incorrect Component	13-6
Add a Component to Content Pane	13-7
Drag and Drop Support	13-7
One Parent for a Component	13-7
JFileChooser Issues with Windows Shortcuts	13-7
nternationalization	

## 14 Internationalization

Troubleshoot Internationalization and Localization	14-1

### 15 Java Sound

Troubleshoot Java Sound Issues

# Part V Submit Bug Reports

## 16 Submit a Bug Report

	Check for Fixes in Update Releases	16-1
	Prepare to Submit a Bug Report	16-1
	Collect Data for a Bug Report	16-2
	Detailed Description of the Problem	16-2
	Hardware Details	16-3
	Operating System Details	16-3
	Java SE Version	16-3
	Command-Line Options	16-4
	Environment Variables	16-4
	Fatal Error Log	16-5
	Core and Crash Dump	16-5
	Logs and Traces	16-5
	Report a Bug	16-5
Collec	Collect Core Dumps	16-5
	Collect Core Dumps on Oracle Solaris	16-6
	Collect Core Dumps on Linux	16-7
	Reasons for Not Getting a Core File	16-8



15-1

# Part VI Appendices

А	Fatal Error Log		
	Location of Fatal Error Log	A-1	
	Description of Fatal Error Log	A-2	
	Header Format	A-2	
	Thread Section Format	A-5	
	Process Section Format	A-8	
	System Section Format	A-14	
В	Java 2D Properties		
	Properties on Oracle Solaris and Linux	B-1	
	Properties on Windows	B-2	
С	Environment Variables and System Properties		
	The JAVA_TOOL_OPTIONS Environment Variable	C-1	
	The java.security.debug System Property	C-2	
D	Command-Line Options		
	Java HotSpot VM Command-Line Options	D-1	
	Other Command-Line Options	D-5	
Е	Summary of Tools in This Release		

# Preface

This document helps you to troubleshoot issues that might occur on the Java Platform, Standard Edition (Java SE) and on Java HotSpot VM. This document provides a description of the available tools and command-line options that can help to analyze problems. This document also provides guidance about debugging core library and client issues and describes some general issues, such as crashes, hangs, and memory leaks. Finally, this document provides directions for data collection and bug report preparation.

# Audience

The target audience for this document is developers who are using the Java Development Kit (JDK), which is Oracle's implementation of Java Platform, Standard Edition (Java SE). Most of the information in this document can be applied to the current and previous releases.

This document is intended for readers with a detailed understanding of the Java Client technologies, a high-level understanding of the components of the Java HotSpot VM, as well as some understanding of concepts such as garbage collection, threads, and native libraries. It is also assumed that the reader is reasonably proficient with the operating system where the Java application is developed and run.

# **Documentation Accessibility**

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

#### Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

# **Related Documents**

For more information about Java SE and the relevant client/desktop technologies, visit Java SE Home.

## Conventions

The following text conventions are used in this document:



Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
monospace	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.



# Part I

# **General Java Troubleshooting**

Java troubleshooting techniques for various diagnostic and monitoring tools, diagnosing memory leaks, and identifying performance issues.

This part describes general Java troubleshooting techniques and contains the following topics.

Prepare Java for Troubleshooting

Provides guidelines for setting up both Java and a Java application for better troubleshooting techniques. These proactive Java setups help debug and narrow down issues with Java and a Java application.

Diagnostic Tools

Describes various diagnostic and monitoring tools used with Java Development Kit (JDK). Further describes the troubleshooting tools available and explains custom tools development using application programming interfaces (APIs).

Troubleshoot Memory Leaks

Provides suggestions for diagnosing problems involving possible memory leaks.

Troubleshoot Performance Issues Using Flight Recorder

Identifies performance issues with a Java application and debugs issues using the Java Flight Recorder.



# 1 Prepare Java for Troubleshooting

This chapter provides some guidelines for setting up both Java and a Java application for better troubleshooting techniques. These proactive Java setups help debug and narrow down issues with Java and the application. Not all suggestions apply to every application. This chapter contains the following sections:

- Set Up Java for Troubleshooting
- Enable Options and Flags for JVM Troubleshooting
- Gather Relevant Data

# Set Up Java for Troubleshooting

Set up the Java environment and command-line options to enable gathering relevant data for troubleshooting.

To set up Java, perform the following:

- 1. Update the Java version: Use the latest Java version to avoid spending time on troubleshooting issues in Java that were fixed. Often, a problem caused by a bug in the Java runtime is fixed in the latest update release. Working with the latest Java version helps avoid some known and common issues.
- 2. Set up the Java environment to debug: Consider the following scenarios while setting up a bigger Java application, starting an application with a launcher script, or running distributed Java on several machines.
  - a. Make it easy to change the Java version: Using the latest Java version helps avoid many runtime issues. If your application starts by running a script, ensure that you have to update the Java path in only one place. If you run in a distributed system, then think about easy ways to change the Java versions across all of the machines.
  - **b.** Make it easy to change the Java command-line options: Sometimes, while troubleshooting, you may want to change Java options; for example, to add a verbose output, to turn off a feature, or to tune Java for better performance. Prepare your systems for these changes.

In a Java application that is running remotely, for example in a testing framework or a cloud solution, ensure that you can still change the Java flags easily. Sometimes, the application takes command-line parameters, or you may want to try a flag quickly to reproduce a problem. Prepare the systems to make these changes easy.

# Enable Options and Flags for JVM Troubleshooting

Set up JVM options and flags to enable gathering relevant data for troubleshooting.

The data you gather depends on the system and what data you would use in case you run into problems. Consider gathering the following data.

 Enable core files: If Java crashes, for example due to a segmentation fault, the OS saves to disk a core file (complete dump of the memory). On Linux and Solaris, core files



are sometimes disabled by default. To enable *core files* on Linux/Solaris, it is usually enough to run the <code>ulimit -c unlimited</code> before starting the application command. Some systems may have different ways to handle these limits.

#### Note:

The core files take up a lot of disk space, especially when run with a large Java heap.

To decide whether to enable *core files*, consider what you would do if you had a crash in your system. Would you want to see a core file? Many Java users won't have much use for a core file. However, if you would want to debug a possible crash either in a native debugger such as gdb or by using the *Serviceability Agent*, then ensure that you enable *core files* before the starting the application.

Many times, crashes are hard to reproduce; therefore, enable core files before the starting the application.

2. Add -XX:+HeapDumpOnOutOfMemoryError to the JVM flags: The -XX:+HeapDumpOnOutOfMemoryError flag saves a Java Heap dump to disk if the applications runs into an OutOfMemoryError.

Like *core files*, heap dumps can be very large, especially when run with a big Java heap.

Again, think about what you would do if the application runs into an OutOfMemoryError. Would you want to inspect the heap at the time of the error? In that case, turn flag by default so that you get this data if the application runs into an unexpected OutOfMemoryError.

3. Run a continuous Java flight recording: Set up Java to run with a continuous flight recording.

Continuous flight recordings are a circular buffer of JFR events. If the application runs into an issue, you can dump the data from the last hour of the run. The JFR events can be helpful to debug a wide range of issues from memory leaks to network errors, high CPU usage, thread blocks, and so on.

The overhead of running with a continuous flight recording is very low. See Produce a Flight Recording for producing a continuous Java Flight Recording.

- 4. Add -verbosegc to the JVM command-line: The flag -verbosegc logs basic information about Java Garbage Collector. This log helps you find the following:
  - Does garbage collection run for a long time?
  - Does the free memory decrease over time?

The garbage collector log helps diagnose issues when the application throws an OutOFMemoryError or the application runs into performance issues; therefore, turning on the -verbosegc flag by default helps troubleshoot issues.



#### Note:

Use log rotation so that an application restart doesn't delete the previous logs. Since JDK7, the flags UseGClogFileRotation and NumberOfGCLogFiles can be used to set up for log rotation. For a description of these flags, see Debugging Options for Java HotSpot VM.

 Print Java version and JVM flags: Before filing a bug on Java or seeking help from a forum, have the basic information handy in the log files. For example, it's helpful to print the Java version and the JVM flags used.

If your application starts with a script, run java -version to print the Java version and print the command line before executing it. Another alternative is to add - XX+PrintCommandLineFlags and -showversion to the JVM arguments.

6. Set up JMC JMX for remote monitoring: JMX can be used to connect to a Java application remotely using tools such as Mission Control or Visual VM. Unless you can run these tools on the same machine that is running your application, setting this up can be helpful later on to monitor the application, send diagnostic commands, manage flight recordings, and so on. There is no performance overhead if you enable JMX.

Another alternative, is to enable JMX after a Java application has started is to use the diagnostic command ManagementAgent.start. Run jcmd <pid> help ManagementAgent.start for a list of flags that can be sent with the command.

See The jcmd Utility.

# Gather Relevant Data

If your application runs into a problem and you want to debug the problem further, ensure that you collect any relevant data before restarting the system, especially if restarting will remove previous files.

- It is important to gather the following files:
  - Core files for crash issues.
  - hs\_err printed text file for Java crashes.
  - Log files: Java and application logs.
  - Java heap dumps for -XX:+HeapDumpOnOutOfMemoryError.
  - Java flight recordings (if enabled). If the problem didn't terminate the application, dump the continuous recordings.
- If the application stopped responding, then gather the following files:
  - Stack traces: Take several stack traces using jcmd <pid> Thread.print before restarting the system.
  - Dump flight recordings (if enabled).
  - Force a core file: If the application can't be closed properly, then stop the application, and force a core file using kill -6 <pid> on Linux or Solaris systems.

### Make a Java Application Easier to Debug

Using a logging framework is a good way to enable future debugging.



If you run into problems in a specific module, you should be able to enable logging in that module. It is also good to specify different levels of logging, for example info, debug, and trace. For more information about Java logging, see Java Logging Overview.

# 2 Diagnostic Tools

The Java Development Kit (JDK) provides diagnostic tools and troubleshooting tools specific to various operating systems. Custom diagnostic tools can also be developed using the APIs provided by the JDK.

This chapter contains the following sections:

- Diagnostic Tools Overview
- JDK Mission Control
- Flight Recorder
- The jcmd Utility
- Native Memory Tracking
- JConsole
- The jdb Utility
- The jinfo Utility
- The jmap Utility
- The jps Utility
- The jstack Utility
- The jstat Utility
- The visualgc Tool
- Control+Break Handler
- Native Operating System Tools
- Custom Diagnostic Tools
- Postmortem Diagnostic Tools
- Hung Processes Tools
- Monitoring Tools
- Other Tools, Options, Variables, and Properties
- The java.lang.management Package
- The java.lang.instrument Package
- The java.lang.Thread Class
- JVM Tool Interface
- The jrunscript Utility
- The jstatd Daemon
- Troubleshooting Tools Based on the Operating System



# **Diagnostic Tools Overview**

Most of the command-line utilities described in this section are either included in the JDK or native operating system tools and utilities.

Although the JDK command-line utilities are included in the JDK download, it is important to consider that they can be used to diagnose issues and monitor applications that are deployed with the Java Runtime Environment (JRE).

In general, the diagnostic tools and options use various mechanisms to get the information they report. The mechanisms are specific to the virtual machine (VM) implementation, operating systems, and release. Frequently, only a subset of the tools is applicable to a given issue at a particular time. Command-line options that are prefixed with -xx are specific to Java HotSpot VM. See Java HotSpot VM Command-Line Options.

#### Note:

The -xx options are not part of the Java API and can vary from one release to the next.

The tools and options are divided into several categories, depending on the type of problem that you are troubleshooting. Certain tools and options might fall into more than one category.

- **Postmortem diagnostics** These tools and options can be used to diagnose a problem after an application crashes. See Postmortem Diagnostic Tools.
- Hung processes These tools can be used to investigate a hung or deadlocked process. See Hung Processes Tools.
- **Monitoring** These tools can be used to monitor a running application. See Monitoring Tools.
- **Other** These tools and options can be used to help diagnose other issues. See Other Tools, Options, Variables, and Properties.

#### Note:

Some command-line utilities described in this section are experimental. The jstack, jinfo, and jmap utilities are examples of utilities that are experimental. It is suggested to use the latest diagnostic utility, jcmd instead of the earlier jstack, jinfo, and jmap utilities.

# JDK Mission Control

Java Platform, Standard Edition (JMC) is a production-time profiling and diagnostics tool. It includes tools to monitor and manage your Java application with very small performance overhead.



JMC's very small performance overhead is a result of its tight integration with the HotSpot VM. JMC functionality is always available on-demand, and its small performance overhead is only in effect while the tools are running. This approach also eliminates the problem of the *observer effect*, which occurs when monitoring tools alter the execution characteristics of the system. JMC enables you to troubleshoot issues and identify root causes and bottlenecks. These properties make the JMC tool ideal for applications running in production.

JMC consists of the following client applications and plug-ins :

- JVM Browser shows running Java applications and their JVMs.
- JMX Console is a mechanism for monitoring and managing JVMs. It connects to a running JVM, collects, displays its characteristics in real time, and enables you to change some of its runtime properties through Managed Beans (MBeans). You can also create rules that trigger on certain events (for example, send an e-mail if the CPU usage by the application reaches 90 percent).
- Flight Recorder (JFR) is a tool for collecting diagnostic and profiling data about a running Java application. It is integrated into the JVM and causes very small performance overhead, so it can be used in production environments. JFR continuously saves large amounts of data about the running applications. This profiling information includes thread samples, lock profiles, and garbage collection details. JFR presents diagnostic information in logically grouped tables and charts. It enables you to select the range of time and level of detail necessary to focus on the problem. Data collected by JFR can be essential when contacting Oracle support to help diagnose issues with your Java application.
- *jcmd Utility or Diagnostic Commands* is used to send diagnostic command requests to the JVM. These requests are useful for managing recordings from Flight Recorder, troubleshooting, and diagnosing JVM and Java applications.
- Plug-ins help in heap dump analysis and DTrace recording. See Plug-in Details. Java SE plug-ins connect to a JVM using the *Java Management Extensions* (JMX) agent. For more information about JMX, see the *Java Platform, Standard Edition Java Management Extensions Guide* .

## Troubleshoot with JDK Mission Control

JMC provides the following features or functionalities that can help you in troubleshooting:

- Java Management console (JMX) connects to a running JVM, and collects and displays key characteristics in real time.
- Triggers user-provided custom actions and rules for JVM.
- Experimental plug-ins from the JMC tool provide troubleshooting activities.
- Flight Recording in JMC is available to analyze events. The preconfigured tabs enable you to easily to drill down in various areas of common interest, such as, code, memory and garbage collection, threads, and I/O. **The Automated Analysis Results** page of flight recordings helps you to diagnose issues quicker. The provided rules and heuristics help you find functional and performance problems in your application and provide tuning tips. Some rules that operate with relatively unknown concepts, like safe points, will provide explanations and links to further information. Some rules are parametrized and can be configured to make more sense in your particular environment. Individual rules can be enabled or disabled as you see fit.
  - Flight Recorder in the JMC application presents diagnostic information in logically grouped tables, charts, and dials. It enables you to select the range of time and level of detail necessary to focus on the problem.



• The JMC plug-ins connect to JVM using the Java Management Extensions (JMX) agent. The JMX is a standard API for the management and monitoring of resources such as applications, devices, services, and the Java Virtual Machine.

# Flight Recorder

Flight Recorder (JFR) is a profiling and event collection framework built into the JDK.

Flight Recorder allows Java administrators and developers to gather detailed low-level information about how a JVM and Java applications are behaving. You can use JMC, with a plug-in, to visualize the data collected by JFR. Flight Recorder and JMC together create a complete toolchain to continuously collect low-level and detailed runtime information enabling after-the-fact incident analysis.

The advantages of using JFR are:

- It records data about JVM events. You can record events at a particular instance of time.
- Recording events with JFR enables you to preserve the execution states to analyze issues. You can access the data anytime to better understand problems and resolve them.
- JFR can record a large amount of data on production systems while keeping the overhead of the recording process low.
- It is most suited for recording latencies. It records situations where the application is not executing as expected and provide details on the bottlenecks.
- It provides insight into how programs interact with execution environment as a whole, ranging from hardware, operating systems, JVM, JDK, and the Java application environment.

Flight recordings can be started when the application is started or while the application is running. The data is recorded as time-stamped data points called *events*. Events are categorized as follows:

- Duration events: occurs at a particular duration with specific start time and stop time.
- Instant events: occurs instantly and gets logged immediately, for example, a thread gets blocked.
- Sample events: occurs at regular intervals to check the overall health of the system, for example, printing heap diagnostics every minute.
- Custom events: user defined events created using JMC or APIs.

In addition, there are predefined events that are enabled in a *recording template*. Some templates only save very basic events and have virtually no impact on performance. Other templates may come with slight performance overhead and may also trigger garbage collections to gather additional data. The following templates are provided with Flight Recorder in the *<JDK ROOT*>/lib/jfr directory:

- default.jfc: Collects a predefined set of data with low overhead.
- profile.jfc: Provides more data than the default.jfc template, but with overhead and impact on performance.

Flight Recorder produces following types of recordings:



• Time fixed recordings: A time fixed recording is also known as a profiling recording that runs for a set amount of time, and then stops. Usually, a time fixed recording has more events enabled and may have a slightly bigger performance effect. Events that are turned on can be modified according to your requirements. Time fixed recordings will be automatically dumped and opened.

Typical use cases for a time fixed recording are as follows:

- Profile which methods are run the most and where most objects are created.
- Look for classes that use more and more heap, which indicates a memory leak.
- Look for bottlenecks due to synchronization and many more such use cases.
- Continuous recordings: A continuous recording is a recording that is always on and saves, for example, the last six hours of data. During this recording, JFR collects events and writes data to the global buffer. When the global buffer fills up, the oldest data is discarded. The data currently in the buffer is written to the specified file whenever you request a dump, or if the dump is triggered by a rule.

A continuous recording with the default template has low overhead and gathers a lot of useful data. However, this template doesn't gather heap statistics or allocation profiling.

## Produce a Flight Recording

The following sections describe different ways to produce a flight recording.

- Start a Flight Recording
- Use Triggers for Automatic Flight Recordings
- Use Startup Flags at the Command Line to Produce a Flight Recording

#### Start a Flight Recording

Follow these steps to start a flight recording using JMC.

- **1.** Find your JVM in the **JVM Browser**.
- 2. Right-click the JVM and select Start Flight Recording...

The Start Flight Recording window opens.

- 3. Click Browse to find a suitable location and file name to save the recording.
- Select either Time fixed recording (profiling recording), or Continuous recording. For continuous recordings, you can specify the maximum size or maximum age of events you want to save.
- Select the flight recording template in the Event settings drop-down list. Templates define the events that you want to record. To create your own templates, click Template Manager. However, for most use cases, select either the Continuous template (for very low overhead recordings) or the Profiling template (for more data and slightly more overhead).
- 6. Click **Finish** to start the recording or click **Next** to modify the event options defined in the selected template.
- Modify the event options for the flight recording. The default settings provide a good balance between data and performance. You can change these settings based on your requirement.

For example:



- The **Threshold** value is the length of event recording. By default, synchronization events above 10 ms are collected. This means, if a thread waits for a lock for more than 10 ms, an event is saved. You can lower this value to get more detailed data for short contentions.
- The **Thread Dump** setting gives you an option to perform periodic thread dumps. These are normal textual thread dumps.
- 8. Click **Finish** to start the recording or click **Next** to modify the event details defined in the selected template.
- 9. Modify the event details for the selected flight recording template. Event details define whether the event should be included in the recording. For some events, you can also define whether a stack trace should be attached to the event, specify the duration threshold (for duration events) and a request period (for requestable events).
- **10.** Click **Back** if you want to modify any of the settings set in the previous steps or click **Finish** to start the recording.

The new flight recording appears in the **Progress View**.

#### Note:

Expand the node in the JVM Browser to view the recordings that are running. Right-click any of the recordings to dump, dump whole, dump last part, edit, stop, or close the recording. Stopping a profiling recording will still produce a recording file and closing a profiling recording will discard the recording.

#### Note:

You can set up JMC to automatically start a flight recording if a condition is met using the **Triggers** tab in the JMX console. For more information, see Use Triggers for Automatic Flight Recordings.

#### Use Triggers for Automatic Flight Recordings

The **Triggers** tab allows you to define and activate rules that trigger events when a certain condition is met. For example, you can set up JDK Mission Control to automatically start a flight recording if a condition is met. This is useful for tracking specific JVM runtime issues.

This is done from the JMX console.

- 1. To start the JMX console, find your application in the JVM Browser, right-click it, and select **Start JMX Console**
- 2. Click the **Triggers** tab at the bottom of the screen.
- 3. Click Add. You can choose any MBean in the application, including your own application-specific ones.

The Add New Rule dialog opens.



- Select an attribute for which the rule should trigger and click Next . For example, select java.lang > OperatingSystem > ProcessCpuLoad.
- 5. Set the condition on which the rule should trigger and click **Next**. For example, set a value for the **Maximum trigger value**, **Sustained period**, and **Limit period**.

#### Note:

You can either select the **Trigger when condition is met** or **Trigger when recovering from condition** check box.

- 6. Select what action you would like your rule to perform when triggered and click Next. For example, choose Start Time Limited Flight Recording and browse the file destination and recording time. Select the Open automatically checkbox, if you wish to open the flight recording automatically when it is triggered.
- 7. Select constraints for your rule and click **Next**. For example, select the particular dates, days of the week, or time of day when the rule should be active.
- 8. Enter a name for your rule and click **Finish**.

The rule is added to the My Rules list.

When you select your rule from the **Trigger Rules** list, the **Rule Details** pane displays its components in the following tabs. You can edit the conditions, attributes, and constraints if you wish:

- Condition
- Action
- Constraint

#### Use Startup Flags at the Command Line to Produce a Flight Recording

Use startup flags to start recording when the application is started. If the application is already running, use the jcmd utility to start recording.

Use the following methods to generate a flight recording:

• Generate a profiling recording when an application is started.

You can configure a time fixed recording at the start of the application using the – XX:StartFlightRecording option. The following example shows how to run the MyApp application and start a 60-second recording 20 seconds after starting the JVM, which will be saved to a file named myrecording.jfr:

```
java -
XX:StartFlightRecording.delay=20s,duration=60s,name=myrecording,filename=myr
ecording.jfr,settings=profile MyApp
```

The settings parameter takes the name of a template. Include the path if the template is not in the *java-home/lib/jfr* directory, which is the location of the default templates. The standard templates are: profile, which gathers more data and is primarily for profiling recordings, and default, which is a low overhead setting made primarily for continuous recordings.

For a complete description of Flight Recorder flags for the java command, see Advanced Runtime Options for Java in the Java Platform, Standard Edition Tools Reference.



• Generate a continuous recording when an application is started.

You can start a continuous recording from the command line using the – XX:StartFlightRecording option. The –XX:FlightRecorderOptions provides additional settings for managing the recording. These flags start a continuous recording that can later be dumped if needed. The following example shows how to run the MyApp application with a continuous recording that saves 6 hours of data to disk. The temporary data will be saved to the /tmp folder.

java -XX:StartFlightRecording.disk=true,maxage=6h,settings=default -XX:FlightRecorderOptions=repository=/tmp MyApp

#### Note:

When you actually dump the recording, you specify a new location for the dumped file, so the files in the repository are only temporary.

· Generate a recording using diagnostic commands.

For a running application, you can generate recordings by using Java commandline diagnostic commands. The simplest way to execute a diagnostic command is to use the jemd tool located in the *java-home/bin* directory. For more details see, The jemd Utility.

The following example shows how to start a recording for the MyApp application with the process ID 5361. 30 minutes of data is recorded and written to /usr/ recording/myapp-recording1.jfr.

```
jcmd 5361 JFR.start duration=30m filename=/usr/recordings/myapp-
recording1.jfr
```

## Analyze a Flight Recording

The following sections describe different ways to analyze a flight recording:

- Analyze a Flight Recording Using JMC
- Analyze a Flight Recording Using the jfr tool or JFR APIs

#### Analyze a Flight Recording Using JMC

Once the flight recording file opens in the JMC, you can look at a number of different areas like code, memory, threads, locks and I/O and analyze various aspects of runtime behavior of your application.

The recording file is automatically opened in the JMC when a timed recording finishes or when a dump of a running recording is created. You can also open any recording file by double-clicking it or by opening it through the **File** menu. The flight recording opens in the **Automated Analysis Results** page. This page helps you to diagnose issues quicker. For example, if you're tuning the garbage collection, or tracking down memory allocation issues, then you can use the memory view to get a detailed view on individual garbage collection events, allocation sites, garbage collection pauses, and so on. You can visualize the latency profile of your application by looking at **I/O** and **Threads** views, and even drill down into a view representing individual events in the recording.



#### View Automated Analysis Results Page

The Flight Recorder extracts and analyzes the data from the recordings and then displays color-coded report logs on the **Automated Analysis Results** page.

By default, results with yellow and red scores are displayed to draw your attention to potential problems. If you want to view all results in the report, click the **Show OK Results** button (a tick mark) on the top-right side of the page. Similarly, to view the results as a table, click the **Table** button.

The benchmarks are mainly divided into problems related to the following:

- Java Application
- JVM Internals
- Environment

Clicking on a heading in the report, for example, **Java Application**, displays a corresponding page.

#### Note:

You can select a respective entry in the **Outline** view to navigate between the pages of the automated analysis.

#### Analyze the Java Application

Java Application dashboard displays the overall health of the Java application.

Concentrate on the parameters having yellow and red scores. The dashboard provides exact references to the problematic situations. Navigate to the specific page to analyze the data and fix the issue.

#### Threads

The **Threads** page provides a snapshot of all the threads that belong to the Java application. It reveals information about an application's thread activity that can help you diagnose problems and optimize application and JVM performance.

Threads are represented in a table and each row has an associated graph. Graphs can help you to identify the problematic execution patterns. The state of each thread is presented as a **Stack Trace**, which provides contextual information of where you can instantly view the problem area. For example, you can easily locate the occurrence of a deadlock.

#### Lock Instances

Lock instances provides further details on threads specifying the lock information, that is, if the thread is trying to take a lock or waiting for a notification on a lock. If a thread has taken any lock, the details are shown in the stack trace.

Memory

One way to detect problems with application performance to is to see how it uses memory during runtime.

In the **Memory** page, the graph represents heap memory usage of the Java application. Each cycle consists of a Java heap growth phase that represents the period of heap memory allocations, followed by a short drop that represents garbage collection, and then the cycle



starts over. The important inference from the graph is that the memory allocations are short-lived as garbage collector pushes down the heap to the start position at each cycle.

Select the **Garbage Collection** check box to see the garbage collection pause time in the graph. It indicates that the garbage collector stopped the application during the pause time to do its work. Long pause times lead to poor application performance, which needs to be addressed.

#### Method Profiling

**Method Profiling** page enables you to see how often a specific method is run and for how long it takes to run a method. The bottlenecks are determined by identifying the methods that take a lot of time to execute.

As profiling generates a lot of data, it is not turned on by default. Start a new recording and select **Profiling - on server** in the **Event settings** drop-down menu. Do a time fixed recording for a short duration. JFR dumps the recording to the file name specified. Open the **Method Profiling** page in JMC to see the top allocations. Top packages and classes are displayed. Verify the details in the stack trace. Inspect the code to verify if the memory allocation is concentrated on a particular object. JFR points to the particular line number where the problem persists.

#### **JVM** Internals

The **JVM Internals** page provides detailed information about the JVM and its behavior.

One of the most important parameters to observe is **Garbage Collections**. Garbage collection is a process of deleting unused objects so that the space can be used for allocation of new objects. The **Garbage Collections** page helps you to better understand the system behavior and garbage collection performance during runtime.

The graphs shows the heap usage as compared to the pause times and how it varies during the specified period. The page also lists all the garbage collection events that occurred during the recording. Observe the longest pause times against the heap. The pause time indicates that garbage collections are taking longer during application processing. It implies that garbage collections are freeing less space on the heap. This situation can lead to memory leaks.

For effective memory management, see the **Compilations** page, which provides details on code compilation along with duration. In large applications, you may have many compiled methods, and memory can be exhausted, resulting in performance issues.

#### Environment

The **Environment** page provides information about the environment in which the recording was made. It helps to understand the CPU usage, memory, and operating system that is being used.

See the **Processes** page to understand concurrent processes running and the competing CPU usage of these processes. The application performance will be affected if many processes use CPU and other system resources.

Check the **Event Browser** page to see the statistics of all the event types. It helps you to focus on the bottlenecks and take appropriate action to improve application performance.



You can create Custom Pages using the **Event Browser** page. Select the required event type from **Event Type Tree** and click the **Create a new page using the select event type** button in the top right corner of the page. The custom page is listed as a new event page below the event browser page.

#### Analyze a Flight Recording Using the jfr tool or JFR APIs

To access the information in a recording from Flight Recorder, use the jfr tool to print event information, or use the Flight Recorder API to programmatically process the data.

Flight Recorder provides the following methods for reviewing the information that was recorded:

- jfr tool Use this command-line tool to print event data from a recording. The tool is located in the java-home/bin directory. For details about this tool, see The jfr Command in the Java Platform, Standard Edition Tools Reference
- Flight Recorder API Use the jdk.jfr.consumer API to extract and format the information in a recording. For more information, see Flight Recorder API Programmer's Guide.

The events in a recording can be used to investigate the following areas:

- General information
  - Number of events recorded at each time stamp
  - Maximum heap usage
  - CPU usage over time, application's CPU usage, and total CPU usage

Watch for CPU usage spiking near 100 percent or the CPU usage is too low or too long garbage collection pauses.

- GC pause time
- JVM information and system properties set
- Memory
  - Memory usage over time

Typically, temporary objects are allocated all the time. When a condition is met, a Garbage Collection (GC) is triggered and all of the objects no longer used are removed. Therefore, the heap usage increases steadily until a GC is triggered, then it drops suddenly. Watch for a steadily increasing heap size over time that could indicate a memory leak.

- Information about garbage collections, including the time spent doing them
- Memory allocations made

The more temporary objects the application allocates, the more the application must perform garbage collection. Reviewing memory allocations helps you find the most allocations and reduce the GC pressure in your application.

Classes that have the most live set

Watch how each object type increases in size during a flight recording. A specific object type that increases a lot in size indicates a memory leak; however, a small variance is normal. Especially, investigate the top growers of non-standard Java classes.

Code



Packages and classes that used the most execution time

Watch where methods are being called from to identify bottlenecks in your application.

- Exceptions thrown
- Methods compiled over time as the application was running
- Number of loaded classes, actual loaded classes and unloaded classes over time
- Threads
  - CPU usage and the number of threads over time
  - Threads that do most of the code execution
  - Objects that are the most waited for due to synchronization
- I/O
  - Information about file reads, file writes, socket reads, and socket writes
- System
  - Information about the CPU, memory and OS of the machine running the application
  - Environment variables and any other processes running at the same time as the JVM
- Events
  - All of the events in the recording

# The jcmd Utility

The jcmd utility is used to send diagnostic command requests to the JVM, where these requests are useful for controlling Java Flight Recordings, troubleshoot, and diagnose JVM and Java applications.

j cmd must be used on the same machine where the JVM is running, and have the same effective user and group identifiers that were used to launch the JVM.

A special command jcmd <process id/main class> PerfCounter.print prints all performance counters in the process.

The command jcmd <process id/main class> <command> [options] sends the command to the JVM.

The following example shows diagnostic command requests to the JVM using j cmd utility.

```
> jcmd
5485 jdk.jcmd/sun.tools.jcmd.JCmd
2125 MyProgram
> jcmd MyProgram (or "jcmd 2125")
2125:
The following commands are available:
Compiler.CodeHeap_Analytics
Compiler.codecache
```



Compiler.codelist Compiler.directives add Compiler.directives clear Compiler.directives print Compiler.directives remove Compiler.queue GC.class histogram GC.class stats GC.finalizer info GC.heap dump GC.heap info GC.run GC.run finalization JFR.check JFR.configure JFR.dump JFR.start JFR.stop JVMTI.agent load JVMTI.data dump ManagementAgent.start ManagementAgent.start local ManagementAgent.status ManagementAgent.stop Thread.print VM.class hierarchy VM.classloader stats VM.classloaders VM.command line VM.dynlibs VM.events VM.flags VM.info VM.log VM.metaspace VM.native memory VM.print touched methods VM.set flag VM.stringtable VM.symboltable VM.system properties VM.systemdictionary VM.uptime VM.version help For more information about a specific command use 'help <command>'. > jcmd MyProgram help Thread.print 2125: Thread.print Print all threads with stacktraces.

Impact: Medium: Depends on the number of threads.



The following sections describe some useful commands and trouble shooting techniques with the j cmd utility:

- Useful Commands for the jcmd Utility
- Troubleshoot with the jcmd Utility

### Useful Commands for the jcmd Utility

The available diagnostic command may be different in different versions of HotSpot VM; therefore, using jcmd <process id/main class> help is the best way to see all available options.

The following are some of the most useful commands in the jcmd tool. Remember you can always use jcmd <process id/main class> help <command> to get any additional options to these commands:

Print full HotSpot and JDK version ID.

jcmd <process id/main class> VM.version

Print all the system properties set for a VM.

There can be several hundred lines of information displayed.

jcmd <process id/main class> VM.system\_properties

• Print all the flags used for a VM.

Even if you have provided no flags, some of the default values will be printed, for example initial and maximum heap size.

jcmd <process id/main class> VM.flags

• Print the uptime in seconds.

jcmd <process id/main class> VM.uptime

• Create a class histogram.

The results can be rather verbose, so you can redirect the output to a file. Both internal and application-specific classes are included in the list. Classes taking the most memory are listed at the top, and classes are listed in a descending order.



jcmd <process id/main class> GC.class histogram

Create a heap dump.

jcmd GC.heap dump filename=Myheapdump

This is the same as using jmap -dump:file=<file> <pid>, but jcmd is the recommended tool to use.

Create a heap histogram.

jcmd <process id/main class> GC.class\_histogram filename=Myheaphistogram

This is the same as using jmap -histo <pid>, but jcmd is the recommended tool to use.

• Print all threads with stack traces.

jcmd <process id/main class> Thread.print

#### Troubleshoot with the jcmd Utility

Use the jcmd utility to troubleshoot.

The jcmd utility provides the following troubleshooting options:

Start a recording.

For example, to start a 2-minute recording on the running Java process with the identifier 7060 and save it to myrecording.jfr in the current directory, use the following:

jcmd 7060 JFR.start name=MyRecording settings=profile delay=20s duration=2m filename=C:\TEMP\myrecording.jfr

Check a recording.

The JFR.check diagnostic command checks a running recording. For example:

jcmd 7060 JFR.check

• Stop a recording.

The JFR.stop diagnostic command stops a running recording and has the option to discard the recording data. For example:

jcmd 7060 JFR.stop

• Dump a recording.

The JFR.dump diagnostic command stops a running recording and has the option to dump recordings to a file. For example:

jcmd 7060 JFR.dump name=MyRecording filename=C:\TEMP\myrecording.jfr

Create a heap dump.

The preferred way to create a heap dump is

jcmd <pid> GC.heap\_dump filename=Myheapdump

• Create a heap histogram.

The preferred way to create a heap histogram is

jcmd <pid> GC.class\_histogram filename=Myheaphistogram



# Native Memory Tracking

The Native Memory Tracking (NMT) is a Java HotSpot VM feature that tracks internal memory usage for a Java HotSpot VM.

Since NMT doesn't track memory allocations by non-JVM code, you may have to use tools supported by the operating system to detect memory leaks in native code.

The following sections describe how to monitor VM internal memory allocations and diagnose VM memory leaks.

- Use NMT to Detect a Memory Leak
- How to Monitor VM Internal Memory
- NMT Memory Categories

## Use NMT to Detect a Memory Leak

Procedure to use Native Memory Tracking to detect memory leaks.

Follow these steps to detect a memory leak:

- Start the JVM with summary or detail tracking using the command line option: -XX:NativeMemoryTracking=summary Or -XX:NativeMemoryTracking=detail.
- Establish an early baseline. Use NMT baseline feature to get a baseline to compare during development and maintenance by running: jcmd <pid>
   VM.native\_memory baseline.
- 3. Monitor memory changes using: jcmd <pid> VM.native memory detail.diff.
- 4. If the application leaks a small amount of memory, then it may take a while to show up.

#### How to Monitor VM Internal Memory

Native Memory Tracking can be set up to monitor memory and ensure that an application does not start to use increasing amounts of memory during development or maintenance.

See Table 2-1 for details about NMT memory categories.

The following sections describe how to get *summary* or *detail* data for NMT and describes how to interpret the sample output.

Interpret sample output: From the following sample output, you will see reserved and committed memory. Note that only committed memory is actually used. For example, if you run with -Xms100m -Xmx1000m, then the JVM will reserve 1000 MB for the Java heap. Because the initial heap size is only 100 MB, only 100 MB will be committed to begin with. For a 64-bit machine where address space is almost unlimited, there is no problem if a JVM reserves a lot of memory. The problem arises if more and more memory gets committed, which may lead to swapping or native out of memory (OOM) situations.

An arena is a chunk of memory allocated using malloc. Memory is freed from these chunks in bulk, when exiting a scope or leaving an area of code. These chunks can be reused in other subsystems to hold temporary memory, for



example, pre-thread allocations. An arena malloc policy ensures no memory leakage. So arena is tracked as a whole and not individual objects. Some initial memory cannot be tracked.

Enabling NMT will result in a **5-10 percent** JVM performance drop, and memory usage for NMT adds 2 machine words to all malloc memory as a malloc header. NMT memory usage is also tracked by NMT.

```
>jcmd 17320 VM.native memory
Native Memory Tracking:
Total: reserved=5699702KB, committed=351098KB
                  Java Heap (reserved=4153344KB, committed=260096KB)
                             (mmap: reserved=4153344KB, committed=260096KB)
                      Class (reserved=1069839KB, committed=22543KB)
                             (classes #3554)
                             ( instance classes #3294, array classes #260)
                             (malloc=783KB #7965)
                             (mmap: reserved=1069056KB, committed=21760KB)
                             ( Metadata:
                                           )
                                 reserved=20480KB, committed=18944KB)
                             (
                                 used=18267KB)
                             (
                                 free=677KB)
                             (
                                 waste=0KB =0.00%)
                             (
                             ( Class space:)
                                 reserved=1048576KB, committed=2816KB)
                             (
                                 used=2454KB)
                             (
                                  free=362KB)
                             (
                                  waste=0KB =0.00%)
                             (
                     Thread (reserved=24685KB, committed=1205KB)
                             (thread #24)
                             (stack: reserved=24576KB, committed=1096KB)
                             (malloc=78KB #132)
                             (arena=30KB #46)
                       Code (reserved=248022KB, committed=7890KB)
                             (malloc=278KB #1887)
                             (mmap: reserved=247744KB, committed=7612KB)
                         GC (reserved=197237KB, committed=52789KB)
                             (malloc=9717KB #2877)
                             (mmap: reserved=187520KB, committed=43072KB)
                   Compiler (reserved=148KB, committed=148KB)
                             (malloc=19KB #95)
                             (arena=129KB #5)
                   Internal (reserved=735KB, committed=735KB)
                             (malloc=663KB #1914)
                             (mmap: reserved=72KB, committed=72KB)
                      Other (reserved=48KB, committed=48KB)
                             (malloc=48KB #4)
```



-	Symbol	(reserved=4835KB, committed=4835KB) (malloc=2749KB #17135) (arena=2086KB #1)
-	Native Memory Tracking	<pre>(reserved=539KB, committed=539KB) (malloc=8KB #109) (tracking overhead=530KB)</pre>
-	Arena Chunk	(reserved=187KB, committed=187KB) (malloc=187KB)
-	Logging	<pre>(reserved=4KB, committed=4KB) (malloc=4KB #179)</pre>
-	Arguments	<pre>(reserved=18KB, committed=18KB) (malloc=18KB #467)</pre>
-	Module	(reserved=62KB, committed=62KB) (malloc=62KB #1060)

 Get detail data: To get a more detailed view of native memory usage, start the JVM with command line option: -XX:NativeMemoryTracking=detail. This will track exactly what methods allocate the most memory. Enabling NMT will result in 5-10 percent JVM performance drop and memory usage for NMT adds 2 words to all malloc memory as malloc header. NMT memory usage is also tracked by NMT.

The following example shows a sample output for virtual memory for track level set to *detail*. One way to get this sample output is to run: jcmd <pid> VM.native\_memory detail.

```
Virtual memory map:
[0x000000702800000 - 0x00000080000000] reserved 4153344KB for
Java Heap from
    [0x00007ffdca6b217d]
    [0x00007ffdca6b19a3]
    [0x00007ffdca6b0d63]
    [0x00007ffdca68e7ae]
        [0x000000702800000 - 0x000000712600000] committed
260096KB from
            [0x00007ffdca254ecc]
            [0x00007ffdca254d52]
            [0x00007ffdca25a5c6]
            [0x00007ffdca2a66bf]
[0x00000080000000 - 0x000000840000000] reserved 1048576KB for
Class from
    [0x00007ffdca6b154a]
    [0x00007ffdca6b0fb4]
    [0x00007ffdca51d2f9]
    [0x00007ffdca51e4d2]
```

[0x00000080000000 - 0x0000008003a0000] committed 3712KB

```
from
            [0x00007ffdca6b10c4]
            [0x00007ffdca6b1250]
            [0x00007ffdca6b0087]
            [0x00007ffdca6af852]
[0x000000bae6d00000 - 0x000000bae6e00000] reserved 1024KB for Thread
Stack from
    [0x00007ffdca679569]
    [0x00007ffdca5751c2]
    [0x00007ffe13ed1ffa]
    [0x00007ffe17d17974]
        [0x000000bae6d00000 - 0x000000bae6d04000] committed 16KB from
            [0x00007ffdca67354e]
            [0x00007ffdca679571]
            [0x00007ffdca5751c2]
            [0x00007ffe13ed1ffa]
        [0x000000bae6df2000 - 0x000000bae6e00000] committed 56KB
[0x000000bae6f00000 - 0x000000bae7000000] reserved 1024KB for Thread
Stack from
    [0x00007ffdca679569]
    [0x00007ffdca5751c2]
    [0x00007ffe13ed1ffa]
    [0x00007ffe17d17974]
        [0x00000bae6f00000 - 0x00000bae6f04000] committed 16KB from
            [0x00007ffdca67354e]
            [0x00007ffdca679571]
            [0x00007ffdca5751c2]
            [0x00007ffe13ed1ffa]
        [0x000000bae6ff3000 - 0x000000bae7000000] committed 52KB
[0x000001d4d3480000 - 0x000001d4d3482000] reserved and committed 8KB for
Internal from
    [0x00007ffdca5df383]
    [0x00007ffdca6737a9]
    [0x00007ffdca322e1d]
    [0x00007ffdca3251e1]
        [0x000001d4d3480000 - 0x000001d4d3482000] committed 8KB from
            [0x00007ffdca5df39b]
            [0x00007ffdca6737a9]
            [0x00007ffdca322e1d]
            [0x00007ffdca3251e1]
[0x000001d4d4d50000 - 0x000001d4d4d60000] reserved and committed 64KB for
Internal from
    [0x00007ffdca5a0719]
    [0x00007ffdca59f627]
    [0x00007ffdca59f03e]
```



```
[0x00007ffdca2b3632]
[0x000001d4d4d60000 - 0x000001d4d4d70000] reserved 64KB for Code
from
    [0x00007ffdca6b159d]
    [0x00007ffdca6b0f40]
    [0x00007ffdca29a2b2]
    [0x00007ffdca1358e9]
        [0x000001d4d4d60000 - 0x000001d4d4d65000] committed 20KB
from
            [0x00007ffdca6b10c4]
            [0x00007ffdca6b1250]
            [0x00007ffdca6b1720]
            [0x00007ffdca29a2ee]
[0x000001d4d51d0000 - 0x000001d4d52c0000] reserved 960KB for Code
from
    [0x00007ffdca6b159d]
    [0x00007ffdca6b0f40]
    [0x00007ffdca29a2b2]
    [0x00007ffdca1358e9]
        [0x000001d4d51d0000 - 0x000001d4d51d5000] committed 20KB
from
            [0x00007ffdca6b10c4]
            [0x00007ffdca6b1250]
            [0x00007ffdca6b1720]
            [0x00007ffdca29a2ee]
        [0x000001d4d51d5000 - 0x000001d4d51e2000] committed 52KB
from
            [0x00007ffdca6b10c4]
            [0x00007ffdca6b1250]
            [0x00007ffdca299df8]
            [0x00007ffdca135acf]
   . . .
[0x000001d4d57f0000 - 0x000001d4d5880000] reserved and committed
576KB for GC from
    [0x00007ffdca24258b]
    [0x00007ffdca2654bb]
    [0x00007ffdca232bcd]
    [0x00007ffdca68d437]
        [0x000001d4d57f0000 - 0x000001d4d5880000] committed 576KB
from
            [0x00007ffdca2425d1]
            [0x00007ffdca2654bb]
            [0x00007ffdca232bcd]
            [0x00007ffdca68d437]
   . . .
[0x000001d4f8930000 - 0x000001d4f9130000] reserved and committed
8192KB for Class from
    [0x00007ffdca6b159d]
```

```
[0x00007ffdca6b0fb4]
    [0x00007ffdca6afee3]
    [0x00007ffdca6af7b5]
        [0x000001d4f8930000 - 0x000001d4f9130000] committed 8192KB from
            [0x00007ffdca6b10c4]
            [0x00007ffdca6b1250]
            [0x00007ffdca6b0087]
            [0x00007ffdca6af852]
[0x000001d4fa0d0000 - 0x000001d4fa2d0000] reserved and committed 2048KB
for Class from
    [0x00007ffdca6b159d]
    [0x00007ffdca6b0fb4]
    [0x00007ffdca6afee3]
    [0x00007ffdca6af7b5]
        [0x000001d4fa0d0000 - 0x000001d4fa2d0000] committed 2048KB from
            [0x00007ffdca6b10c4]
            [0x00007ffdca6b1250]
            [0x00007ffdca6b0087]
            [0x00007ffdca6af852]
   . . .
```

• Get diff from NMT baseline: For both *summary* and *detail* level tracking, you can set a baseline after the application is up and running. Do this by running jcmd <pid> VM.native\_memory baseline after the application warms up. Then, you can runjcmd <pid> VM.native\_memory summary.diff or jcmd <pid> VM.native\_memory detail.diff.

The following example shows sample output for the *summary* difference in native memory usage since the baseline was set and is a great way to find memory leaks.

```
>jcmd 17320 VM.native memory summary.diff
17320:
Total: reserved=5712754KB +8236KB, committed=370550KB +12940KB
                  Java Heap (reserved=4153344KB, committed=260096KB)
                            (mmap: reserved=4153344KB, committed=260096KB)
                      Class (reserved=1078291KB +6357KB,
committed=32915KB +7381KB)
                            (classes #4868 +958)
                            ( instance classes #4528 +901, array classes
#340 +57)
                            (malloc=1043KB +213KB #12345 +3198)
                            (mmap: reserved=1077248KB +6144KB,
committed=31872KB +7168KB)
                            ( Metadata:
                                          )
                            (
                                 reserved=28672KB +6144KB,
committed=27904KB +6400KB)
                                 used=27206KB +6181KB)
                            (
                                 free=698KB +219KB)
                            (
```



```
waste=0KB =0.00%)
                            (
                             ( Class space:)
                                 reserved=1048576KB,
                             (
committed=3968KB +768KB)
                             (
                                 used=3395KB +643KB)
                                 free=573KB +125KB)
                             (
                                 waste=0KB =0.00%)
                             (
                     Thread (reserved=26745KB +2KB,
committed=1421KB +6KB)
                            (thread #26)
                             (stack: reserved=26624KB,
committed=1300KB +4KB)
                             (malloc=85KB #142)
                             (arena=35KB +2 #50)
                       Code (reserved=248533KB +323KB,
committed=14725KB +3999KB)
                            (malloc=789KB +323KB #4505 +1596)
                             (mmap: reserved=247744KB,
committed=13936KB +3676KB)
                         GC (reserved=197345KB +70KB,
committed=52897KB +70KB)
                             (malloc=9825KB +70KB #4868 +1395)
                             (mmap: reserved=187520KB,
committed=43072KB)
                   Compiler (reserved=153KB +4KB, committed=153KB
+4KB)
                             (malloc=27KB +6KB #312 +154)
                             (arena=126KB -2 #5)
                   Internal (reserved=785KB +27KB, committed=785KB
+27KB)
                             (malloc=713KB +27KB #2213 +214)
                             (mmap: reserved=72KB, committed=72KB)
                      Other (reserved=49KB, committed=49KB)
                            (malloc=49KB #4)
                     Symbol (reserved=6268KB +1082KB,
committed=6268KB +1082KB)
                             (malloc=3926KB +1018KB #34608 +16640)
                            (arena=2342KB +64 #1)
    Native Memory Tracking (reserved=963KB +364KB, committed=963KB
+364KB)
                             (malloc=9KB +1KB #123 +8)
                             (tracking overhead=953KB +363KB)
                Arena Chunk (reserved=187KB, committed=187KB)
                            (malloc=187KB)
                    Logging (reserved=4KB, committed=4KB)
```

(malloc=4KB #179)

\_

\_

Arguments (reserved=18KB, committed=18KB) (malloc=18KB #467) Module (reserved=71KB +7KB, committed=71KB +7KB) (malloc=71KB +7KB #1119 +53)

The following example is a sample output that shows the *detail* difference in native memory usage since the baseline and is a great way to find memory leaks.

 [0x00007ffdca51ce00] [0x00007ffdca127ca3] [0x00007ffdca51d08b] [0x00007ffdca195288]	(malloc=81KB type=Class +18KB #869 +194)
[0x00007ffdca169f01] [0x00007ffdca16480a] [0x00007ffdca164349] [0x00007ffdca16444d]	(malloc=3KB type=Compiler +1KB #27 +8)
[0x00007ffdca5c160a] [0x000001d4ddd73b66]	(malloc=2KB type=GC +2KB #1 +1)
[0x00007ffdca5c160a] [0x00007ffdca22d16b] [0x00007ffdca254a62] [0x00007ffdca264b9e]	(malloc=6KB type=GC +6KB #3 +3)
 [0x00007ffdca2b860a] [0x00007ffdca166d7c] [0x00007ffdca3237bf] [0x00007ffdca313331]	(malloc=16KB type=Class +1KB #61 +6)
[0x00007ffdca67170c] [0x00007ffdca6712f3] [0x00007ffdca369ed1] [0x000001d4ddd6f0b7]	(malloc=3KB type=Internal +1KB #9 +3)
[0x00007ffdca60a90c] [0x00007ffdca60ca3f] [0x00007ffdca60cd29] [0x00007ffdca2d78f3]	(malloc=16KB type=Symbol +6KB #1030 +399)
[0x00007ffdca60a90c]	

```
[0x00007ffdca60ca3f]
[0x00007ffdca60cc2e]
[0x00007ffdca19a631]
                              (malloc=116KB type=Symbol +23KB #7411
+1442)
   . . .
[0x00007ffdca29860f]
[0x00007ffdca204dc4]
[0x00007ffdca65070a]
[0x00007ffdca64fd17]
                              (malloc=11KB type=Class +3KB #357 +82)
[0x00007ffdca29860f]
[0x00007ffdca204dc4]
[0x00007ffdca65070a]
[0x00007ffdca64aefb]
                              (malloc=105KB type=Class +23KB #3371
+749)
[0x00007ffdca50c00f]
[0x00007ffdca50be9d]
[0x00007ffdca552fc9]
[0x00007ffdca203aa0]
                              (malloc=1KB type=Native Memory
Tracking +1KB #20 +20)
[0x00007ffdca53dd17]
[0x00007ffdca53f52a]
[0x00007ffdca350c54]
[0x00001d4ddd6f0b7]
   . . .
```

### NMT Memory Categories

List of native memory tracking memory categories used by NMT.

Table 2-1 describes native memory categories used by NMT. These categories may change with a release.

Category	Description
Java Heap	The heap where your objects live
Class	Class meta data
Thread	Memory used by threads, including thread data structure, resource area, handle area, and so on
Code	Generated code
GC	Data use by the GC, such as card table
Compiler	Memory tracking used by the compiler when generating code
Internal	Memory that does not fit the previous categories, such as the memory used by the command line parser, JVMTI, properties, and so on

Table 2-1 Native Memory Tracking Memory Categories



Category	Description
Other	Memory not covered by another category
Symbol	Memory for symbols
Native Memory Tracking	Memory used by NMT
Arena Chunk	Memory used by chunks in the arena chunk pool
Logging	Memory used by logging
Arguments	Memory for arguments
Module	Memory used by modules

 Table 2-1 (Cont.) Native Memory Tracking Memory Categories

### JConsole

Another useful tool included in the JDK download is the JConsole monitoring tool. This tool is compliant with JMX. The tool uses the built-in JMX instrumentation in the JVM to provide information about the performance and resource consumption of running applications.

Although the tool is included in the JDK download, it can also be used to monitor and manage applications deployed with the JRE.

The JConsole tool can attach to any Java application in order to display useful information such as thread usage, memory consumption, and details about class loading, runtime compilation, and the operating system.

This output helps with the high-level diagnosis of problems such as memory leaks, excessive class loading, and running threads. It can also be useful for tuning and heap sizing.

In addition to monitoring, JConsole can be used to dynamically change several parameters in the running system. For example, the setting of the -verbose:gc option can be changed so that the garbage collection trace output can be dynamically enabled or disabled for a running application.

The following sections describe troubleshooting techniques with the JConsole tool.

- Troubleshoot with the JConsole Tool
- Monitor Local and Remote Applications with JConsole

#### Troubleshoot with the JConsole Tool

Use the JConsole tool to monitor data.

The following list provides an idea of the data that can be monitored using the JConsole tool. Each heading corresponds to a tab pane in the tool.

Overview

This pane displays graphs that shows the heap memory usage, number of threads, number of classes, and CPU usage over time. This overview allows you to visualize the activity of several resources at once.

- Memory
  - For a selected memory area (heap, non-heap, various memory pools):



- \* Graph showing memory usage over time
- \* Current memory size
- \* Amount of committed memory
- \* Maximum memory size
- Garbage collector information, including the number of collections performed, and the total time spent performing garbage collection
- Graph showing the percentage of heap and non-heap memory currently used

In addition, on this pane you can request garbage collection to be performed.

- Threads
  - Graph showing thread usage over time.
  - Live threads: Current number of live threads.
  - Peak: Highest number of live threads since the JVM started.
  - For a selected thread, the name, state, and stack trace, as well as, for a blocked thread, the synchronizer that the thread is waiting to acquire, and the thread that owns the lock.
  - The **Deadlock Detection** button sends a request to the target application to perform deadlock detection and displays each deadlock cycle in a separate tab.
- Classes
  - Graph showing the number of loaded classes over time
  - Number of classes currently loaded into memory
  - Total number of classes loaded into memory since the JVM started, including those subsequently unloaded
  - Total number of classes unloaded from memory since the JVM started
- VM Summary
  - General information, such as the JConsole connection data, uptime for the JVM, CPU time consumed by the JVM, compiler name, total compile time, and so on.
  - Thread and class summary information
  - Memory and garbage collection information, including number of objects pending finalization, and so on
  - Information about the operating system, including physical characteristics, the amount of virtual memory for the running process, and swap space
  - Information about the JVM itself, such as the arguments and class path
- MBeans

This pane displays a tree structure that shows all platform and application MBeans that are registered in the connected JMX agent. When you select an MBean in the tree, its attributes, operations, notifications, and other information are displayed.

 You can invoke operations, if any. For example, the operation dumpHeap for the HotSpotDiagnostic MBean, which is in the com.sun.management domain, performs a heap dump. The input parameter for this operation is the path name of the heap dump file on the machine where the target VM is running.



- You can set the value of writable attributes. For example, you can set, unset, or change the value of certain VM flags by invoking the setVMOption operation of the HotSpotDiagnostic MBean. The flags are indicated by the list of values of the DiagnosticOptions attribute.
- You can subscribe to notifications, if any, by using the Subscribe and Unsubscribe buttons.

### Monitor Local and Remote Applications with JConsole

JConsole can monitor both local applications and remote applications. If you start the tool with an argument specifying a JMX agent to connect to, then the tool will automatically start monitoring the specified application.

To monitor a local application, execute the command jconsolepid, where pid is the process ID of the application.

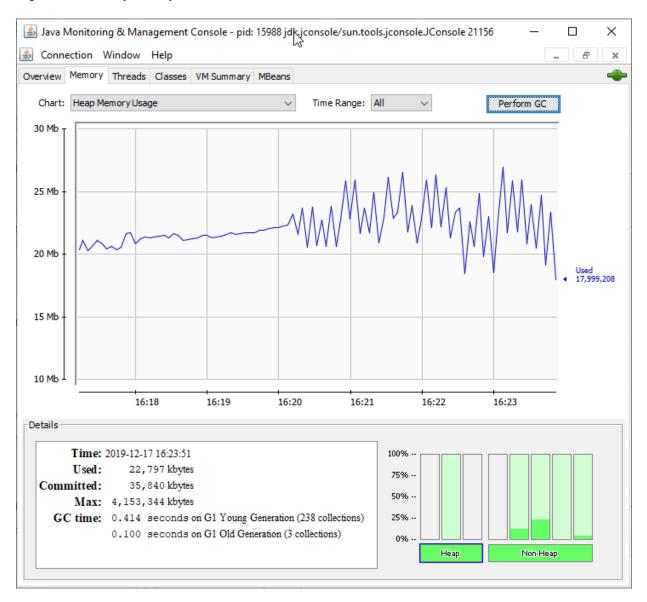
To monitor a remote application, execute the command jconsolehostname: portnumber, where hostname is the name of the host running the application, and portnumber is the port number you specified when you enabled the JMX agent.

If you execute the jconsole command without arguments, the tool will start by displaying the **New Connection** window, where you specify the local or remote process to be monitored. You can connect to a different host at any time by using the **Connection** menu.

With the latest JDK releases, no option is necessary when you start the application to be monitored.

As an example of the output of the monitoring tool, Figure 2-1 shows a chart of the heap memory usage.





#### Figure 2-1 Sample Output from JConsole

## The jdb Utility

The jdb utility is included in the JDK as an example command-line debugger. The jdb utility uses the Java Debug Interface (JDI) to launch or connect to the target JVM.

The JDI is a high-level Java API that provides information useful for debuggers and similar systems that need access to the running state of a (usually remote) virtual machine. JDI is a component of the Java Platform Debugger Architecture (JPDA). See Java Platform Debugger Architecture.

The following section provides troubleshooting techniques for the jdb utility.

Troubleshoot with the jdb Utility



### Troubleshoot with the jdb Utility

The jdb utility is used to monitor the debugger connectors used for remote debugging.

In JDI, a connector is the way that the debugger connects to the target JVM. The JDK traditionally ships with connectors that launch and establish a debugging session with a target JVM, as well as connectors that are used for remote debugging (using TCP/IP or shared memory transports).

These connectors are generally used with enterprise debuggers, such as the NetBeans integrated development environment (IDE) or commercial IDEs.

The command jdb -listconnectors prints a list of the available connectors. The command jdb -help prints the command usage help.

See jdb Utility in the Java Platform, Standard Edition Tools Reference

## The jinfo Utility

The jinfo command-line utility gets configuration information from a running Java process or crash dump, and prints the system properties or the command-line flags that were used to start the JVM.

Java Mission Control, Java Flight Recorder, and jcmd utility can be used for diagnosing problems with JVM and Java applications. Use the latest utility, jcmd, instead of the previous jinfo utility for enhanced diagnostics and reduced performance overhead.

With the -flag option, the jinfo utility can dynamically set, unset, or change the value of certain JVM flags for the specified Java process. See Java HotSpot VM Command-Line Options.

The output for the jinfo utility for a Java process with PID number 29620 is shown in the following example.

```
c:\Program Files\Java\jdk-11\bin>jinfo 29620
Java System Properties:
sun.desktop=windows
awt.toolkit=sun.awt.windows.WToolkit
java.specification.version=11
sun.cpu.isalist=amd64
sun.jnu.encoding=Cp1252
java.class.path=C\:\\sampleApps\\DynamicTreeDemo\\dist\\DynamicTreeDemo.jar
sun.awt.enableExtraMouseButtons=true
java.vm.vendor=Oracle Corporation
sun.arch.data.model=64
user.variant=
java.vendor.url=http\://java.oracle.com/
user.timezone=
java.vm.specification.version=11
os.name=Windows 10
sun.java.launcher=SUN STANDARD
user.country=US
sun.boot.library.path=c\:\\Program Files\\Java\\jdk-11.0.5\\bin
```



```
sun.java.command=C\:\\sampleApps\\DynamicTreeDemo\\dist\
\DynamicTreeDemo.jar
jdk.debug=release
sun.cpu.endian=little
user.home=C\:\\Users\\user1
user.language=en
sun.stderr.encoding=cp437
java.specification.vendor=Oracle Corporation
java.version.date=2019-10-15
java.home=c\:\\Program Files\\Java\\jdk-11.0.5
file.separator=\\
java.vm.compressedOopsMode=Zero based
line.separator=\r\n
sun.stdout.encoding=cp437
java.specification.name=Java Platform API Specification
java.vm.specification.vendor=Oracle Corporation
java.awt.graphicsenv=sun.awt.Win32GraphicsEnvironment
user.script=
sun.management.compiler=HotSpot 64-Bit Tiered Compilers
java.runtime.version=11.0.5+10-LTS
user.name=user1
path.separator=;
os.version=10.0
java.runtime.name=Java(TM) SE Runtime Environment
file.encoding=Cp1252
java.vm.name=Java HotSpot(TM) 64-Bit Server VM
java.vendor.version=18.9
java.vendor.url.bug=http\://bugreport.java.com/bugreport/
java.io.tmpdir=C\:\\Users\\user1\\AppData\\Local\\Temp\\
java.version=11.0.5
user.dir=c\:\\Users\\user1
os.arch=amd64
java.vm.specification.name=Java Virtual Machine Specification
java.awt.printerjob=sun.awt.windows.WPrinterJob
sun.os.patch.level=
java.library.path=c\:\\Program Files\\Java\\jdk-11.0.5\\bin;...
java.vendor=Oracle Corporation
java.vm.info=mixed mode
java.vm.version=11.0.5+10-LTS
sun.io.unicode.encoding=UnicodeLittle
java.class.version=55.0
VM Flags:
   . . .
```

The following topic describes the troubleshooting technique with jinfo utility.

Troubleshooting with the jinfo Utility

### Troubleshooting with the jinfo Utility

The output from jinfo provides the settings for java.class.path and sun.boot.class.path.

If you start the target JVM with the -classpath and -Xbootclasspath arguments, then the output from jinfo provides the settings for java.class.path and sun.boot.class.path. This information might be needed when investigating class loader issues.

In addition to getting information from a process, the jhsdb jinfo tool can use a core file as input. On the Oracle Solaris operating system, for example, the gcore utility can be used to get a core file of the process in the preceding example. The core file will be named core.29620 and will be generated in the working directory of the process. The path to the Java executable file and the core file must be specified as arguments to the jhsdb jinfo utility, as shown in the following example.

```
$ jhsdb jinfo --exe java-home/bin/java --core core.29620
```

Sometimes, the binary name will not be java. This happens when the VM is created using the JNI invocation API. The jhsdb jinfo tool requires the binary from which the core file was generated.

## The jmap Utility

The jmap command-line utility prints memory-related statistics for a running VM or core file. For a core file, use jhsdb jmap.

JDK Mission Control, Flight Recorder, and jcmd utility can be used for diagnosing problems with JVM and Java applications. It is suggested to use the latest utility, jcmd instead of the previous jmap utility for enhanced diagnostics and reduced performance overhead.

If jmap is used with a process or core file without any command-line options, then it prints the list of shared objects loaded (the output is similar to the pmap utility on Oracle Solaris operating system). For more specific information, you can use the options -heap, -histo, or - clstats. These options are described in the subsections that follow.

In addition, the JDK 7 release introduced the -dump:format=b,file=filename option, which causes jmap to dump the Java heap in binary format to a specified file.

If the jmap *pid* command does not respond because of a hung process, then use the jhsdb jmap utility to run the Serviceability Agent.

The following sections describe troubleshooting techniques with examples that print memoryrelated statistics for a running VM or a core file.

- Heap Configuration and Usage
- Heap Histogram
- Class Loader Statistics



#### Heap Configuration and Usage

Use the jhsdb jmap --heap command to get the Java heap information.

The --heap option is used to get the following Java heap information:

- Information specific to the garbage collection (GC) algorithm, including the name
  of the GC algorithm (for example, parallel GC) and algorithm-specific details (such
  as the number of threads for parallel GC).
- Heap configuration that might have been specified as command-line options or selected by the VM based on the machine configuration.
- Heap usage summary: For each generation (area of the heap), the tool prints the total heap capacity, in-use memory, and available free memory. If a generation is organized as a collection of spaces (for example, the new generation), then a space-specific memory size summary is included.

The following example shows output from the jhsdb jmap --heap command.

```
$ jhsdb jmap --heap --pid 29620
Attaching to process ID 29620, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 11.0.5+10-LTS
using thread-local object allocation.
Garbage-First (G1) GC with 4 thread(s)
Heap Configuration:
  MinHeapFreeRatio= 40MaxHeapFreeRatio= 70MaxHeapSize= 42
                           = 4253024256 (4056.0MB)
   NewSize
                           = 1363144 (1.2999954223632812MB)
                       = 2551185408 (2433.0MB)
= 5452592 (5.1999969482421875MB)
   MaxNewSize
   OldSize
   NewRatio
                           = 2

        SurvivorRatio
        - 2

        MetaspaceSize
        = 8

   CompressedClassSpaceSize = 1073741824 (1024.0MB)
   MaxMetaspaceSize = 17592186044415 MB
   G1HeapRegionSize
                        = 1048576 (1.0 MB)
Heap Usage:
G1 Heap:
   regions = 4056
   capacity = 4253024256 (4056.0MB)
   used = 10485760 (10.0MB)
   free = 4242538496 (4046.0MB)
   0.2465483234714004% used
G1 Young Generation:
Eden Space:
   regions = 11
   capacity = 15728640 (15.0MB)
   used = 11534336 (11.0MB)
```



```
free
         = 4194304 (4.0 MB)
  73.33333333333333% used
Survivor Space:
  regions = 0
  capacity = 0 (0.0MB)
  used = 0 (0.0MB)
         = 0 (0.0MB)
  free
  0.0% used
G1 Old Generation:
  regions = 0
  capacity = 250609664 (239.0MB)
  used = 0 (0.0MB)
  free = 250609664 (239.0MB)
  0.0% used
```

#### Heap Histogram

The jmap command with the -histo option or the jhsdb jmap --histo command can be used to get a class-specific histogram of the heap.

The jmap -histo command can print the heap histogram for a running process. Use jhsdb jmap --histo to print the heap histogram for a core file.

When the jmap -histo command is executed on a running process, the tool prints the number of objects, memory size in bytes, and fully qualified class name for each class. Internal classes in the Java HotSpot VM are enclosed within angle brackets. The histogram is useful to understand how the heap is used. To get the size of an object, you must divide the total size by the count of that object type.

The following example shows output from the jmap -histo command when it is executed on a process with PID number 29620.

```
$ jmap -histo 29620
 num #instances
                                #bytes class name (module)
_____
                37127
                               2944304 [B (java.base@11)
   1:

        5773
        1860840
        [I (java.base@11)

        15844
        887264
        jdk.internal.org.objectweb.asm.Item

   2:
   3:
(java.base@11)
   4: 24061 577464 java.lang.String (java.base@11)
5: 13334 575120 [Ljava.lang.Object; (java.base@11)
                 562
                               373280 [Ljdk.internal.org.objectweb.asm.Item;
   6:
(java.base@11)

      7:
      2575
      313392
      java.lang.Class (java.base@11)

      8:
      8233
      250792
      [Ljava.lang.Class; (java.base@

      9:
      6043
      241720
      java.lang.invoke.MethodType

                               250792 [Ljava.lang.Class; (java.base@11)
(java.base@11)
                6716
  10:
                                214912
java.lang.invoke.MethodType$ConcurrentWeakInternSet$WeakEntry (java.base@11)
         6324 202368 java.util.HashMap$Node (java.base@11)
  11:
                5352
  12:
                           171264 java.lang.invoke.LambdaForm$Name
(java.base@11)
                               155160 [C (java.base@11)
  13:
                612
  14:
                  594
                                 133056
jdk.internal.org.objectweb.asm.MethodWriter (java.base@11)
```



```
15:
              1538
                         110864
[Ljava.lang.invoke.LambdaForm$Name; (java.base@11)
             4521 108504 java.lang.StringBuilder
 16:
(java.base@11)
 17:
            2252 108096 java.lang.invoke.MemberName
(java.base@11)
 18:
              644
                          103208 [Ljava.util.HashMap$Node;
(java.base@11)
 19:
             1375
                           77000
java.lang.invoke.LambdaFormEditor$Transform (java.base@11)
 20:
              2215
                           70880
java.util.concurrent.ConcurrentHashMap$Node (java.base@11)
... more lines removed here to reduce output...
1425:
                1
                              16
sun.util.resources.LocaleData$LocaleDataStrategy (java.base@11)
1426:
                1
                              16
sun.util.resources.provider.NonBaseLocaleDataMetaInfo
(jdk.localedata@11)
Total
           184008
                        11075800
```

When the jhsdb jmap -histo command is executed on a core file, the tool prints serial number, number of instances, bytes, and class name for each class. Internal classes in the Java HotSpot VM are prefixed with an asterisk (\*).

The following example shows output of the <code>jhsdb jmap -histo</code> command when it is executed on a core file.

```
& jhsdb jmap --exe /usr/java/jdk_11/bin/java --core core.16395 --
histoDebugger attached successfully.
Server compiler detected.
JVM version is 11.0.5+10-LTS
Iterating over heap. This may take a while...
Object Histogram:
```

num	#instances	#bytes	Class description
1:	11102	564520	byte[]
2:	10065	241560	java.lang.String
3:	1421	163392	java.lang.Class
4:	26403	2997816	* ConstMethodKlass
5:	26403	2118728	* MethodKlass
6:	39750	1613184	* SymbolKlass
7:	2011	1268896	* ConstantPoolKlass
8:	2011	1097040	* InstanceKlassKlass
9:	1906	882048	* ConstantPoolCacheKlass
10:	1614	125752	java.lang.Object[]
11:	1160	64960	jdk.internal.org.objectweb.asm.Item
12:	1834	58688	java.util.HashMap\$Node
13:	359	40880	java.util.HashMap\$Node[]
14:	1189	38048	
java.ut	il.concurrent.	.Concurrent	HashMap\$Node
15:	46	37280	jdk.internal.org.objectweb.asm.Item[]
16:	29	35600	char[]
17:	968	32320	int[]



18:	650	26000	java.lang.invoke.MethodType
19:	475	22800	java.lang.invoke.MemberName

### **Class Loader Statistics**

Use the  $\verb"jmap"$  command with the <code>-clstats</code> option to print class loader statistics for the Java heap.

The jmap command connects to a running process using the process ID and prints detailed information about classes loaded in the Metaspace:

- Index Unique index for the class
- Super Index number of the super class
- InstBytes Number of bytes per instance
- KlassBytes Number of bytes for the class
- annotations Size of annotations
- CpAll Combined size of the constants, tags, cache, and operands per class
- MethodCount Number of methods per class
- Bytecodes Number of bytes used for byte codes
- MethodAll Combined size of the bytes per method, CONSTMETHOD, stack map, and method data
- ROAII Size of class metadata that could be put in read-only memory
- RWAII Size of class metadata that must be put in read/write memory
- Total Sum of ROAll + RWAll
- ClassName Name of the loaded class

The following example shows a subset of the output from the jmap -clstats command when it is executed on a process with PID number 10952.

c:\Progr	am Fil	es\Java\j	dk-11.0.5	\bin>jm	ap -cl	stats 10	)952	
Index Su	per In	stBytes K	lassBytes	annota	tions	CpAll	MethodCount	Bytecodes
MethodAl	l RO	All RWA	.ll Tot	al Clas	sName			
1	-1	304816	512		0	0	0	
0	0	24	624	648 [B				
2	51	285264	784		0	23344	147	
5815	48848	28960	46640	75600	java.	lang.Cla	ASS	
3	-1	256368	512		0	0	0	
0	0	24	624	648 [I				
4	51	166344	680		136	17032	123	
5433	48256	23920	44160	68080	java.	lang.Str	ring	
5	-1	146360	512		0	0	0	
0	0	24	624	648 [L	java.]	.ang.Obje	ect;	
6	51	123680	600		0	1384	7	
149	1888	1200	3024	4224	java.u	util.Hash	nMap\$Node	
7	51	52928	608		0	1360	9	
213	2472	1632	3184	4816				
java.uti	l.conc	urrent.Co	ncurrentH	ashMap\$	Node			
8	-1	51888	512	-	0	0	0	
0	0	24	624	648 [C				



### The jps Utility

The jps utility lists every instrumented Java HotSpot VM for the current user on the target system.

The utility is very useful in environments where the VM is embedded, that is, where it is started using the JNI Invocation API rather than the java launcher. In these environments, it is not always easy to recognize the Java processes in the process list.

The following example shows the use of the jps utility.

```
$ jps
16217 MyApplication
16342 jps
```

The jps utility lists the virtual machines for which the user has access rights. This is determined by access-control mechanisms specific to the operating system. On the Oracle Solaris operating system, for example, if a non-root user executes the jps utility, then the output is a list of the virtual machines that were started with that user's UID.

In addition to listing the PID, the utility provides options to output the arguments passed to the application's main method, the complete list of VM arguments, and the



full package name of the application's main class. The jps utility can also list processes on a remote system if the remote system is running the jstatd daemon.

# The jstack Utility

Use the jcmd or jhsdb jstack utility, instead of the jstack utility to diagnose problems with JVM and Java applications.

JDK Mission Control, Flight Recorder, and jcmd utility can be used to diagnose problems with JVM and Java applications. It is suggested to use the latest utility, jcmd, instead of the previous jstack utility for enhanced diagnostics and reduced performance overhead.

The following sections describe troubleshooting techniques with the jstack and jhsdb jstack utilities.

- Troubleshoot with the jstack Utility
- Stack Trace from a Core Dump
- Mixed Stack

### Troubleshoot with the jstack Utility

The jstack command-line utility attaches to the specified process, and prints the stack traces of all threads that are attached to the virtual machine, including Java threads and VM internal threads, and optionally native stack frames. The utility also performs deadlock detection. For core files, use jhsdb jstack.

A stack trace of all threads can be useful in diagnosing a number of issues, such as deadlocks or hangs.

The -1 option instructs the utility to look for ownable synchronizers in the heap and print information about java.util.concurrent.locks. Without this option, the thread dump includes information only on monitors.

The output from the jstack *pid* option is the same as that obtained by pressing Ctrl+\ at the application console (standard input) or by sending the process a quit signal. See Control+Break Handler for an example of the output.

Thread dumps can also be obtained programmatically using the Thread.getAllStackTraces method, or in the debugger using the debugger option to print all thread stacks (the where command in the case of the jdb sample debugger).

### Stack Trace from a Core Dump

Use the jhsdb jstack command to obtain stack traces from a core dump.

To get stack traces from a core dump, execute the jhsdb jstack command on a core file, as shown in the following example.

```
$ jhsdb jstack --exe java-home/bin/java --core core-file
```



#### Mixed Stack

The jhsdb jstack utility can also be used to print a mixed stack; that is, it can print native stack frames in addition to the Java stack. Native frames are the C/C++ frames associated with VM code and JNI/native code.

To print a mixed stack, use the --mixed option, as shown in the following example.

```
>jhsdb jstack --mixed --pid 21177
Attaching to process ID 21177, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 11.0.5+10-LTS
Deadlock Detection:
No deadlocks found.
----- 0 ------
0x00007ffe17e8f7e4 ntdll!ZwWaitForSingleObject + 0x14
----- 1 -----
----- 2 ------
"DestroyJavaVM" #19 prio=5 tid=0x000001a5607af000 nid=0x5ad8 waiting
on condition [0x00000000000000]
  java.lang.Thread.State: RUNNABLE
  JavaThread state: thread blocked
                  ntdll!ZwWaitForSingleObject + 0x14
0x00007ffe17e8f7e4
----- 3 ------
0x00007ffe17e8f7e4
                   ntdll!ZwWaitForSingleObject + 0x14
_____ 4 _____
0x00007ffe17e8f7e4
                  ntdll!ZwWaitForSingleObject + 0x14
----- 5 -----
0x00007ffe17e8f7e4
                   ntdll!ZwWaitForSingleObject + 0x14
----- 6 -----
0x00007ffe17e8f7e4
                   ntdll!ZwWaitForSingleObject + 0x14
----- 7 ------
0x00007ffe17e8f7e4
                  ntdll!ZwWaitForSingleObject + 0x14
----- 8 -----
0x00007ffe17e8f7e4 ntdll!ZwWaitForSingleObject + 0x14
----- 9 -----
"Reference Handler" #2 daemon prio=10 tid=0x000001a57f747800
nid=0x2ecc waiting on condition [0x0000060f3afe000]
  java.lang.Thread.State: RUNNABLE
  JavaThread state: thread blocked
0x00007ffe17e8f7e4 ntdll!ZwWaitForSingleObject + 0x14
----- 10 ------
"Finalizer" #3 daemon prio=8 tid=0x000001a50400c000 nid=0x3e70 in
Object.wait() [0x0000060f3bfe000]
  java.lang.Thread.State: WAITING (on object monitor)
  JavaThread state: thread blocked
0x00007ffe17e8f7e4 ntdll!ZwWaitForSingleObject + 0x14
----- 11 ------
"Signal Dispatcher" #4 daemon prio=9 tid=0x000001a504062800 nid=0x550
runnable [0x00000000000000]
  java.lang.Thread.State: RUNNABLE
```

```
JavaThread state: thread blocked
0x00007ffe17e8f7e4 ntdll!ZwWaitForSingleObject + 0x14
----- 12 ------
"Attach Listener" #5 daemon prio=5 tid=0x000001a504063800 nid=0x488c
runnable [0x00000000000000]
  java.lang.Thread.State: RUNNABLE
  JavaThread state: thread blocked
0x00007ffe17e8f7e4 ntdll!ZwWaitForSingleObject + 0x14
0x000001a504064340
                            ?????????
----- 13 -----
"C2 CompilerThread0" #6 daemon prio=9 tid=0x000001a504066000 nid=0x5968
waiting on condition [0x0000000000000]
  java.lang.Thread.State: RUNNABLE
  JavaThread state: thread blocked
0x00007ffe17e8f7e4 ntdll!ZwWaitForSingleObject + 0x14
0x0400030091000000
                            ----- 14 ------
"C1 CompilerThread0" #8 daemon prio=9 tid=0x000001a50406d800 nid=0x67c
waiting on condition [0x0000000000000]
  java.lang.Thread.State: RUNNABLE
  JavaThread state: thread blocked
0x00007ffe17e8f7e4 ntdll!ZwWaitForSingleObject + 0x14
----- 15 -----
"Sweeper thread" #9 daemon prio=9 tid=0x000001a50406e800 nid=0x4690 runnable
[0x00000000000000000]
  java.lang.Thread.State: RUNNABLE
  JavaThread state: thread blocked
0x00007ffe17e8f7e4 ntdll!ZwWaitForSingleObject + 0x14
0x010a012700000004
                           ----- 16 -----
"Service Thread" #10 daemon prio=9 tid=0x000001a5041fd800 nid=0x3060
runnable [0x00000000000000]
  java.lang.Thread.State: RUNNABLE
  JavaThread state: thread blocked
                 ntdll!ZwWaitForSingleObject + 0x14
0x00007ffe17e8f7e4
----- 17 ------
0x00007ffe17e8f7e4 ntdll!ZwWaitForSingleObject + 0x14
----- 18 -----
"Common-Cleaner" #11 daemon prio=8 tid=0x000001a504205800 nid=0x5db4 in
Object.wait() [0x0000060f43ff000]
  java.lang.Thread.State: TIMED WAITING (on object monitor)
  JavaThread state: thread blocked
                  ntdll!ZwWaitForSingleObject + 0x14
0x00007ffe17e8f7e4
----- 19 -----
"Java2D Disposer" #12 daemon prio=10 tid=0x000001a50c8ef800 nid=0x58e8 in
Object.wait() [0x0000060f44fe000]
  java.lang.Thread.State: WAITING (on object monitor)
  JavaThread state: thread blocked
0x00007ffe17e8f7e4 ntdll!ZwWaitForSingleObject + 0x14
----- 20 -----
"AWT-Shutdown" #13 prio=5 tid=0x000001a50c8d0800 nid=0x3a34 in Object.wait()
[0x0000060f45ff000]
  java.lang.Thread.State: WAITING (on object monitor)
  JavaThread state: thread blocked
0x00007ffe17e8f7e4 ntdll!ZwWaitForSingleObject + 0x14
```



Frames that are prefixed with an asterisk (\*) are Java frames, whereas frames that are not prefixed with an asterisk are native C/C++ frames.

The output of the utility can be piped through c++filt to demangle C++ mangled symbol names. Because the Java HotSpot VM is developed in the C++ language, the jhsdb jstack utility prints C++ mangled symbol names for the Java HotSpot internal functions.

The c++filt utility is delivered with the native C++ compiler suite: SUNWspro on the Oracle Solaris operating system and gnu on Linux.

### The jstat Utility

The jstat utility uses the built-in instrumentation in the Java HotSpot VM to provide information about performance and resource consumption of running applications.

The tool can be used when diagnosing performance issues, and in particular issues related to heap sizing and garbage collection. The jstat utility does not require the VM to be started with any special options. The built-in instrumentation in the Java HotSpot VM is enabled by default. This utility is included in the JDK download for all operating system platforms supported by Oracle.

#### Note:

The instrumentation is not accessible on a FAT32 file system.

See jstat in the Java Platform, Standard Edition Tools Reference.

The jstat utility uses the virtual machine identifier (VMID) to identify the target process. The documentation describes the syntax of the VMID, but its only required component is the local virtual machine identifier (LVMID). The LVMID is typically (but not always) the operating system's PID for the target JVM process.

The jstat utility provides data similar to the data provided by the vmstat and iostat on Oracle Solaris and Linux operating systems.

For a graphical representation of the data, you can use the visualgc tool. See The visualgc Tool.



The following example illustrates the use of the <code>-gcutil</code> option, where the <code>jstat</code> utility attaches to LVMID number 2834 and takes 7 samples at 250-millisecond intervals.

#### \$ jstat -gcutil 2834 250 7

SO	S1	Ε	0	М	YGC	YGCT	FGC	FGCT	GCT
0.00	99.74	13.49	7.86	95.82	3	0.124	0	0.000	0.124
0.00	99.74	13.49	7.86	95.82	3	0.124	0	0.000	0.124
0.00	99.74	13.49	7.86	95.82	3	0.124	0	0.000	0.124
0.00	99.74	13.49	7.86	95.82	3	0.124	0	0.000	0.124
0.00	99.74	13.49	7.86	95.82	3	0.124	0	0.000	0.124
0.00	99.74	13.49	7.86	95.82	3	0.124	0	0.000	0.124
0.00	99.74	13.49	7.86	95.82	3	0.124	0	0.000	0.124

The output of this example shows you that a young generation collection occurred between the third and fourth samples. The collection took 0.017 seconds and promoted objects from the eden space (E) to the old space (O), resulting in an increase of old space utilization from 46.56% to 54.60%.

The following example illustrates the use of the <code>-gcnew</code> option where the <code>jstat</code> utility attaches to LVMID number 2834, takes samples at 250-millisecond intervals, and displays the output. In addition, it uses the <code>-h3</code> option to display the column headers after every 3 lines of data.

#### \$ jstat -gcnew -h3 2834 250

1 1000	- yene.								
SOC	S1C	SOU	S1U T	T MTT	DSS	EC	EU	YGC	YGCT
192.0	192.0	0.0	0.0	15 15	96.0	1984.0	942.0	218	1.999
192.0	192.0	0.0	0.0	15 15	96.0	1984.0	1024.8	218	1.999
192.0	192.0	0.0	0.0	15 15	96.0	1984.0	1068.1	218	1.999
SOC	S1C	SOU	S1U	TT MTT	DSS	EC	EU	YGC	YGCT
192.0	192.0	0.0	0.0	15 15	96.0	1984.0	1109.0	218	1.999
192.0	192.0	0.0	103.2	1 15	96.0	1984.0	0.0	219	2.019
192.0	192.0	0.0	103.2	1 15	96.0	1984.0	71.6	219	2.019
SOC	S1C	SOU	S1U	TT MTT	DSS	EC	EU	YGC	YGCT
192.0	192.0	0.0	103.2	1 15	96.0	1984.0	73.7	219	2.019
192.0	192.0	0.0	103.2	1 15	96.0	1984.0	78.0	219	2.019
192.0	192.0	0.0	103.2	1 15	96.0	1984.0	116.1	219	2.019

In addition to showing the repeating header string, this example shows that between the fourth and fifth samples, a young generation collection occurred, whose duration was 0.02 seconds. The collection found enough live data that the survivor space 1 utilization (S1U) would have exceeded the desired survivor size (DSS). As a result, objects were promoted to the old generation (not visible in this output), and the tenuring threshold (TT) was lowered from 15 to 1.

The following example illustrates the use of the <code>-gcoldcapacity</code> option, where the <code>jstat</code> utility attaches to LVMID number 21891 and takes 3 samples at 250-millisecond intervals. The <code>-t</code> option is used to generate a time stamp for each sample in the first column.

#### \$ jstat -gcoldcapacity -t 21891 250 3

Timestamp	OGCMN	OGCMX	OGC	OC	YGC	FGC	FGCT	GCT
150.1	1408.0	60544.0	11696.0	11696.0	194	80	2.874	3.799
150.4	1408.0	60544.0	13820.0	13820.0	194	81	2.938	3.863
150.7	1408.0	60544.0	13820.0	13820.0	194	81	2.938	3.863



The Timestamp column reports the elapsed time in seconds since the start of the target JVM. In addition, the <code>-gcoldcapacity</code> output shows the old generation capacity (OGC) and the old space capacity (OC) increasing as the heap expands to meet the allocation or promotion demands. The OGC has grown from 11696 KB to 13820 KB after the  $81^{st}$  full generation capacity (FGC). The maximum capacity of the generation (and space) is 60544 KB (OGCMX), so it still has room to expand.

# The visualgc Tool

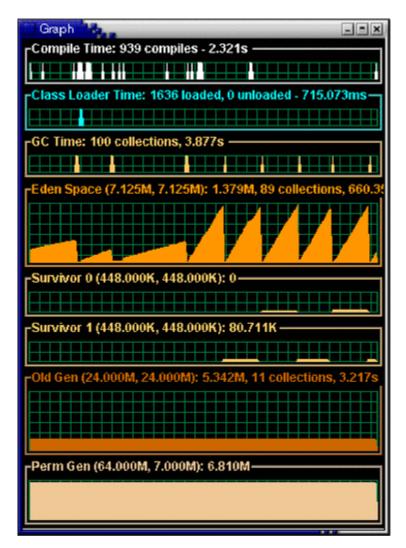
The visualge tool provides a graphical view of the garbage collection (GC) system.

The visualgc tool is related to the jstat tool. See The jstat Utility. The visualgc tool provides a graphical view of the garbage collection (GC) system. As with jstat, it uses the built-in instrumentation of the Java HotSpot VM.

The visualgc tool is not included in the JDK release, but is available as a separate download from the jvmstat technology page.

Figure 2-2 shows how the GC and heap are visualized.

#### Figure 2-2 Sample Output from visualgc



## **Control+Break Handler**

The result of pressing the Control key and the backslash (\) key at the application console on operating systems such as Oracle Solaris or Linux, or Windows.

On Oracle Solaris or Linux operating systems, the combination of pressing the Control key and the backslash () key at the application console (standard input) causes the Java HotSpot VM to print a thread dump to the application's standard output. On Windows, the equivalent key sequence is the Control and Break keys. The general term for these key combinations is the Control+Break handler.

On Oracle Solaris and Linux operating systems, a thread dump is printed if the Java process receives a quit signal. Therefore, the kill -QUIT *pid* command causes the process with the ID *pid* to print a thread dump to standard output.

The following sections describe the data traced by the Control+Break handler:

- Thread Dump
- Thread States for a Thread Dump
- Detect Deadlocks
- Heap Summary

#### Thread Dump

The thread dump consists of the thread stack, including the thread state, for all Java threads in the virtual machine.

The thread dump does not terminate the application: it continues after the thread information is printed.

The following example illustrates a thread dump.

```
Full thread dump Java HotSpot(TM) Client VM (1.6.0-rc-b100 mixed mode):
"DestroyJavaVM" prio=10 tid=0x00030400 nid=0x2 waiting on condition
[0x0000000..0xfe77fbf0]
   java.lang.Thread.State: RUNNABLE
"Thread2" prio=10 tid=0x000d7c00 nid=0xb waiting for monitor entry
[0xf36ff000..0xf36ff8c0]
   java.lang.Thread.State: BLOCKED (on object monitor)
        at Deadlock$DeadlockMakerThread.run(Deadlock.java:32)
        - waiting to lock <0xf819a938> (a java.lang.String)
        - locked <0xf819a970> (a java.lang.String)
"Thread1" prio=10 tid=0x000d6c00 nid=0xa waiting for monitor entry
[0xf37ff000..0xf37ffbc0]
   java.lang.Thread.State: BLOCKED (on object monitor)
        at Deadlock$DeadlockMakerThread.run(Deadlock.java:32)
        - waiting to lock <0xf819a970> (a java.lang.String)
        - locked <0xf819a938> (a java.lang.String)
"Low Memory Detector" daemon prio=10 tid=0x000c7800 nid=0x8 runnable
```



```
[0x0000000..0x0000000]
   java.lang.Thread.State: RUNNABLE
"CompilerThread0" daemon prio=10 tid=0x000c5400 nid=0x7 waiting on
condition [0x0000000..0x0000000]
   java.lang.Thread.State: RUNNABLE
"Signal Dispatcher" daemon prio=10 tid=0x000c4400 nid=0x6 waiting on
condition [0x0000000..0x0000000]
   java.lang.Thread.State: RUNNABLE
"Finalizer" daemon prio=10 tid=0x000b2800 nid=0x5 in Object.wait()
[0xf3f7f000..0xf3f7f9c0]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0xf4000b40> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:116)
        - locked <0xf4000b40> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:132)
        at
java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:159)
"Reference Handler" daemon prio=10 tid=0x000ae000 nid=0x4 in
Object.wait() [0xfe57f000..0xfe57f940]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0xf4000a40> (a java.lang.ref.Reference$Lock)
        at java.lang.Object.wait(Object.java:485)
        at
java.lang.ref.Reference$ReferenceHandler.run(Reference.java:116)
        - locked <0xf4000a40> (a java.lang.ref.Reference$Lock)
"VM Thread" prio=10 tid=0x000ab000 nid=0x3 runnable
"VM Periodic Task Thread" prio=10 tid=0x000c8c00 nid=0x9 waiting on
condition
```

The output consists of a number of thread entries separated by an empty line. The Java Threads (threads that are capable of executing Java language code) are printed first, and these are followed by information about VM internal threads. Each thread entry consists of a header line followed by the thread stack trace.

The header line contains the following information about the thread:

- Thread name.
- Indication if the thread is a daemon thread.
- Thread priority (prio).
- Thread ID (tid), which is the address of a thread structure in memory.
- ID of the native thread (nid).
- Thread state, which indicates what the thread was doing at the time of the thread dump. See Table 2-2 for more details.
- Address range, which gives an estimate of the valid stack region for the thread.



### Thread States for a Thread Dump

List of possible thread states for a thread dump.

Table 2-2 lists the possible thread states for a thread dump using the Control+Break Handler.

Table 2-2 Thread States for a Thread Dump

Thread State	Description
NEW	The thread has not yet started.
RUNNABLE	The thread is executing in the JVM.
BLOCKED	The thread is blocked, waiting for a monitor lock.
WAITING	The thread is waiting indefinitely for another thread to perform a particular action.
TIMED_WAITING	The thread is waiting for another thread to perform an action for up to a specified waiting time.
TERMINATED	The thread has exited.

#### Detect Deadlocks

The Control+Break handler can be used to detect deadlocks in threads.

In addition to the thread stacks, the Control+Break handler executes a deadlock detection algorithm. If any deadlocks are detected, then the Control+Break handler, as shown in the following example, prints additional information after the thread dump about each deadlocked thread.

```
Found one Java-level deadlock:
_____
"Thread2":
 waiting to lock monitor 0x000af330 (object 0xf819a938, a java.lang.String),
 which is held by "Thread1"
"Thread1":
 waiting to lock monitor 0x000af398 (object 0xf819a970, a java.lang.String),
 which is held by "Thread2"
Java stack information for the threads listed above:
_____
"Thread2":
       at Deadlock$DeadlockMakerThread.run(Deadlock.java:32)
       - waiting to lock <0xf819a938> (a java.lang.String)
       - locked <0xf819a970> (a java.lang.String)
"Thread1":
       at Deadlock$DeadlockMakerThread.run(Deadlock.java:32)
       - waiting to lock <0xf819a970> (a java.lang.String)
       - locked <0xf819a938> (a java.lang.String)
```

Found 1 deadlock.

If the JVM flag -XX:+PrintConcurrentLocks is set, then the Control+Break handler will also print the list of concurrent locks owned by each thread.



#### Heap Summary

The Control+Break handler can be used to print a heap summary.

The following example shows the different generations (areas of the heap), with the size, the amount used, and the address range. The address range is especially useful if you are also examining the process with tools such as pmap.

```
Неар
def new generation total 1152K, used 435K [0x22960000, 0x22a90000,
0x22e40000
)
 eden space 1088K, 40% used [0x22960000, 0x229ccd40, 0x22a70000)
 from space 64K, 0% used [0x22a70000, 0x22a70000, 0x22a80000)
 to space 64K, 0% used [0x22a80000, 0x22a80000, 0x22a90000)
tenured generation total 13728K, used 6971K [0x22e40000,
0x23ba8000, 0x269600
00)
  the space 13728K, 50% used [0x22e40000, 0x2350ecb0, 0x2350ee00,
0x23ba8000)
compacting perm gen total 12288K, used 1417K [0x26960000,
0x27560000, 0x2a9600
(0.0)
  the space 12288K, 11% used [0x26960000, 0x26ac24f8, 0x26ac2600,
0x27560000)
   ro space 8192K, 62% used [0x2a960000, 0x2ae5ba98, 0x2ae5bc00,
0x2b160000)
   rw space 12288K, 52% used [0x2b160000, 0x2b79e410, 0x2b79e600,
0x2bd60000)
```

If the JVM flag -XX:+PrintClassHistogram is set, then the Control+Break handler will produce a heap histogram.

## Native Operating System Tools

List of native tools available on Windows, Linux, and Oracle Solaris operating systems that are useful for troubleshooting or monitoring purposes.

A brief description is provided for each tool. For further details, see the operating system documentation (or man pages for the Oracle Solaris and Linux operating systems).

The format of log files and output from command-line utilities depends on the release. For example, if you develop a script that relies on the format of the fatal error log, then the same script may not work if the format of the log file changes in a future release.

You can also search for Windows-specific debug support on the MSDN developer network.

The following sections describe troubleshooting techniques and improvements to a few native operating system tools.

- Troubleshooting Tools Based on the Operating System
- DTrace Tool



- Probe Providers in Java HotSpot VM
- Improvements to the pmap Utility
- Improvements to the pstack Utility

### **DTrace Tool**

The Oracle Solaris 10 operating system includes the DTrace tool, which allows dynamic tracing of the operating system kernel and user-level programs.

This tool supports scripting at system-call entry and exit, at user-mode function entry and exit, and at many other probe points. The scripts are written in the **D programming language**, which is a C-like language with safe pointer semantics. These scripts can help you to troubleshoot problems or solve performance issues.

The dtrace command is a generic front end to the DTrace tool. This command provides a simple interface to invoke the D language, to retrieve buffered trace data, and to access a set of basic routines to format and print traced data.

You can write your own customized DTrace scripts, using the D language, or download and use one or more of the many scripts that are already available on various sites.

The probes are delivered and instrumented by kernel modules called providers. The types of tracing offered by the probe providers include user instruction tracing, function boundary tracing, kernel lock instrumentation, profile interrupt, system call tracing, and many more. If you write your own scripts, you use the D language to enable the probes; this language also allows conditional tracing and output formatting.

You can use the dtrace -1 command to explore the set of providers and probes that are available on your Oracle Solaris operating system.

The DTraceToolkit is a collection of useful documented scripts developed by the Open Oracle Solaris DTrace community. See DTraceToolkit.

See Solaris Dynamic Tracing Guide.

### Probe Providers in Java HotSpot VM

The Java HotSpot VM contains two built-in probe providers hotspot and hotspot jni.

These providers deliver probes that can be used to monitor the internal state and activities of the VM, as well as the Java application that is running.

The JVM probe providers can be categorized as follows:

- VM lifecycle: VM initialization begin and end, and VM shutdown
- Thread lifecycle: thread start and stop, thread name, thread ID, and so on
- Class-loading: Java class loading and unloading
- Garbage collection: Start and stop of garbage collection, systemwide or by memory pool
- Method compilation: Method compilation begin and end, and method loading and unloading
- Monitor probes: Wait events, notification events, contended monitor entry and exit
- Application tracking: Method entry and return, allocation of a Java object



In order to call from native code to Java code, the native code must make a call through the JNI interface. The <code>hotspot\_jni</code> provider manages DTrace probes at the entry point and return point for each of the methods that the JNI interface provides for invoking Java code and examining the state of the VM.

At probe points, you can print the stack trace of the current thread using the ustack built-in function. This function prints Java method names in addition to C/C++ native function names. The following example is a simple D script that prints a full stack trace whenever a thread calls the read system call.

```
#!/usr/sbin/dtrace -s
syscall::read:entry
/pid == $1 && tid == 1/ {
    ustack(50, 0x2000);
}
```

The script in the previous example is stored in a file named read.d and is run by specifying the PID of the Java process that is traced as shown in the following example.

```
read.d pid
```

If your Java application generated a lot of I/O or had some unexpected latency, then the DTrace tool and its ustack() action can help you to diagnose the problem.

#### Improvements to the pmap Utility

Improvements to the pmap utility in Oracle Solaris 10 operating system.

The pmap utility was improved in Oracle Solaris 10 operating system to print stack segments with the text [stack]. This text helps you to locate the stack easily.

The following example shows the stack trace with improved pmap utility.

```
19846:
         /net/myserver/export1/user/j2sdk6/bin/java -Djava.endorsed.d
00010000
            72K r-x-- /export/disk09/jdk/6/rc/b63/binaries/
solsparc/bin/java
00030000 16K rwx-- /export/disk09/jdk/6/rc/b63/binaries/
solsparc/bin/java
00034000 32544K rwx--
                       [ heap ]
D1378000
           32K rwx-R [ stack tid=44 ]
D1478000
           32K rwx-R [ stack tid=43 ]
D1578000
           32K rwx-R [ stack tid=42 ]
D1678000
           32K rwx-R [ stack tid=41 ]
           32K rwx-R
D1778000
                        [ stack tid=40 ]
          32K rwx-R [stack tid=39]
D1878000
D1974000
           48K rwx-R [ stack tid=38 ]
           32K rwx-R [ stack tid=37 ]
D1A78000
          32K rwx-R
D1B78000
                       [ stack tid=36 ]
[.. more lines removed here to reduce output ..]
FF370000 8K r-x-- /usr/lib/libsched.so.1
FF380000
            8K r-x-- /platform/sun4u-us3/lib/libc psr.so.1
FF390000
            16K r-x-- /lib/libthread.so.1
```



FF3A4000	8K	rwx	/lib/libthread.so.1
FF3B0000	8K	r-x	/lib/libdl.so.1
FF3C0000	168K	r-x	/lib/ld.so.1
FF3F8000	8K	rwx	/lib/ld.so.1
FF3FA000	8K	rwx	/lib/ld.so.1
FFB80000	24K		[ anon ]
FFBF0000	64K	rwx	[ stack ]
total	167224K		

### Improvements to the pstack Utility

Improvements to the pstack utility in Oracle Solaris 10 operating system.

Before Oracle Solaris 10 operating system, the pstack utility did not support Java. It printed hexadecimal addresses for both interpreted and compiled Java methods.

Starting with Oracle Solaris 10 operating system, the pstack command-line tool prints mixedmode stack traces (Java and C/C++ frames) from a core file or a live process. The utility prints Java method names for interpreted, compiled, and inlined Java methods.

# **Custom Diagnostic Tools**

The JDK has extensive APIs to develop custom tools to observe, monitor, profile, debug, and diagnose issues in applications that are deployed in the JRE.

The development of new tools is beyond the scope of this document. Instead, this section provides a brief overview of the APIs available.

All the packages mentioned in this section are described in the Java SE API specification.

See the example and demonstration code that is included in the JDK download.

The following sections describe packages, interface classes, and the Java debugger that can be used as custom diagnostic tools for troubleshooting.

- The java.lang.management Package
- The java.lang.instrument Package
- The java.lang.Thread Class
- JVM Tool Interface
- Java Platform Debugger Architecture

### Java Platform Debugger Architecture

The Java Platform Debugger Architecture (JPDA) is the architecture designed for use by debuggers and debugger-like tools.

The Java Platform Debugger Architecture consists of two programming interfaces and a wire protocol:

- The Java Virtual Machine Tool Interface (JVM TI) is the interface to the virtual machine. See JVM Tool Interface.
- The Java Debug Interface (JDI) defines information and requests at the user code level. It is a pure Java programming language interface for debugging Java programming



language applications. In JPDA, the JDI is a remote view in the debugger process of a virtual machine in the process being debugged. It is implemented by the front end, where as a debugger-like application (for example, IDE, debugger, tracer, or monitoring tool) is the client. See the module jdk.jdi.

 The Java Debug Wire Protocol (JDWP) defines the format of information and requests transferred between the process being debugged and the debugger front end, which implements the JDI.

The jdb utility is included in the JDK as an example command-line debugger. The jdb utility uses the JDI to launch or connect to the target VM. See The jdb Utility.

In addition to traditional debugger-type tools, the JDI can also be used to develop tools that help in postmortem diagnostics and scenarios where the tool needs to attach to a process in a noncooperative manner (for example, a hung process).

### **Postmortem Diagnostic Tools**

List of tools and options available for post-mortem diagnostics of problems between the application and the Java HotSpot VM.

Table 2-3 summarizes the options and tools that are designed for postmortem diagnostics. If an application crashes, then these options and tools can be used to get additional information, either at the time of the crash or later using information from the crash dump.

Tool or Option	Description and Usage
Fatal Error Log	When an irrecoverable (fatal) error occurs, an error log is created. This file contains information obtained at the time of the fatal error. In many cases, it is the first item to examine when a crash occurs. See Fatal Error Log.
- XX:+HeapDumpOnOutOfMe moryError <b>option</b>	This command-line option specifies the generation of a heap dump when the VM detects a native out-of-memory error. See The -XX:HeapDumpOnOutOfMemoryError Option.
-XX:OnError option	This command-line option specifies a sequence of user-supplied scripts or commands to be executed when a fatal error occurs. For example, on Windows, this option can execute a command to force a crash dump. This option is very useful on systems where a postmortem debugger is not configured. See The -XX:OnError Option.
- XX:+ShowMessageBoxOnE rror <b>option</b>	This command-line option suspends a process when a fatal error occurs. Depending on the user response, the option can launch the native debugger (for example, dbx, gdb, msdev) to attach to the VM. See The -XX:ShowMessageBoxOnError Option.
Other -XX options	Several other -XX command-line options can be useful in troubleshooting. See Other -XX Options.
jhsdb jinfo <b>utility</b>	This utility can get configuration information from a core file obtained from a crash or from a core file obtained using the gcore utility. See The jinfo Utility.
jhsdb jmap <b>utility</b>	This utility can get memory map information, including a heap histogram, from a core file obtained from a crash or from a core file obtained using the gcore utility. See The jmap Utility.

#### Table 2-3 Postmortem Diagnostics Tools



Tool or Option	Description and Usage
jstack utility	This utility can get Java and native stack information from a Java process. On the Oracle Solaris and Linux operating systems, the utility can also get the information from a core file or a remote debug server. See The jstack Utility.
Native tools	Each operating system has native tools and utilities that can be used for postmortem diagnosis. See Native Operating System Tools.

Table 2-3 (Cont.) Postmortem Diagnostics Tools	Table 2-3	(Cont.)	Postmortem	<b>Diagnostics Tools</b>
--	-----------	---------	------------	--------------------------

### Hung Processes Tools

List of tools and options for diagnosing problems between the application and the Java HotSpot VM in a hung process.

 Table 2-4 summarizes the options and tools that can help in scenarios involving a hung or deadlocked process. These tools do not require any special options to start the application.

Java Mission Control, Java Flight Recorder, and the jemd utility can be used to diagnose problems with JVM and Java applications. It is suggested to use the latest utility, jemd, instead of the previous jstack, jinfo, and jmap utilities for enhanced diagnostics and reduced performance overhead.

Tool or Option	Description and Usage
Ctrl+Break handler (Control+\ or kill -QUIT pid on the Oracle Solaris and Linux operating systems, and Control+Break on Windows)	This key combination performs a thread dump and deadlock detection. The Ctrl+Break handler can optionally print a list of concurrent locks and their owners, as well as a heap histogram. See Control+Break Handler.
jcmd utility	This utility is used to send diagnostic command requests to the JVM, where these requests are useful for controlling Java Flight Recordings (JFRs). The JFRs are used to troubleshoot and diagnose flight recording events. See The jcmd Utility.
jdb utility	Debugger support includes attaching connectors, which allow $jdb$ and other Java language debuggers to attach to a process. This can help show what each thread is doing at the time of a hang or deadlock. See The jdb Utility.
jinfo <b>utility</b>	This utility can get configuration information from a Java process. See The jinfo Utility.
jmap <b>utility</b>	This utility can get memory map information, including a heap histogram, from a Java process. The jhsdb jmap utility can be used if the process is hung. See The jmap Utility.
jstack utility	This utility can obtain Java and native stack information from a Java process. See The jstack Utility.
Native tools	Each operating system has native tools and utilities that can be useful in hang or deadlock situations. See Native Operating System Tools.

#### Table 2-4 Hung ProcessTools



# **Monitoring Tools**

Tools and options for monitoring running applications and detecting problems are available in the JDK and in the operating system.

The tools listed in the Table 2-5 are designed for monitoring applications that are running.

Java Mission Control, Java Flight Recorder, and the jcmd utility can be used to diagnose problems with JVM and Java applications. It is suggested to use the latest utility, jcmd, instead of the previous jstack, jinfo, and jmap utilities for enhanced diagnostics and reduced performance overhead.

#### Table 2-5 Monitoring Tools

Tool or Option	Description and Usage
Java Mission Control	Java Mission Control (JMC) is a new JDK profiling and diagnostic tool platform for HotSpot JVM. It is a tool suite for basic monitoring, managing, and production time profiling and diagnostics with high performance. Java Mission Control minimizes the performance overhead that's usually an issue with profiling tools.
jcmd utility	This utility is used to send diagnostic command requests to the JVM, where these requests are useful for controlling Java Flight Recordings. The JFRs are used to troubleshoot and diagnose JVM and Java applications with flight recording events. See The jcmd Utility.
JConsole utility	This utility is a monitoring tool that is based on Java Management Extensions (JMX). The tool uses the built-in JMX instrumentation in the Java Virtual Machine to provide information about the performance and resource consumption of running applications. See JConsole.
jmap <b>utility</b>	This utility can get memory map information, including a heap histogram, from a Java process or a core file. See The jmap Utility.
jps <b>utility</b>	This utility lists the instrumented Java HotSpot VMs on the target system. The utility is very useful in environments where the VM is embedded, that is, it is started using the JNI Invocation API rather than the java launcher. See The jps Utility.
jstack <b>utility</b>	This utility can get Java and native stack information from a Java process. The utility can also get the information from a core file. See The jstack Utility.
jstat <b>utility</b>	This utility uses the built-in instrumentation in Java to provide information about performance and resource consumption of running applications. The tool can be used when diagnosing performance issues, especially those related to heap sizing and garbage collection. See The jstat Utility.
jstatd <b>daemon</b>	This tool is a Remote Method Invocation (RMI) server application that monitors the creation and termination of instrumented Java Virtual Machines and provides an interface to allow remote monitoring tools to attach to VMs running on the local host. See The jstatd Daemon.
visualgc <b>utility</b>	This utility provides a graphical view of the garbage collection system. As with jstat, it uses the built-in instrumentation of Java HotSpot VM. See The visualgc Tool.
Native tools	Each operating system has native tools and utilities that can be useful for monitoring purposes. For example, the dynamic tracing (DTrace) capability introduced in Oracle Solaris 10 operating system performs advanced monitoring. See Native Operating System Tools.

# Other Tools, Options, Variables, and Properties

List of general troubleshooting tools, options, variables, and properties that can help to diagnose issues.

In addition to the tools that are designed for specific types of problems, the tools, options, variables, and properties listed in Table 2-6 can help in diagnosing other issues.

JDK Mission Control, Flight Recorder, and the jcmd utility can be used for diagnosing problems with JVM and Java applications. It is suggested to use the latest utility, jcmd, instead of the previous jstack, jinfo, and jmap utilities for enhanced diagnostics and reduced performance overhead.

Tool or Option	Description and Usage
JDKMission Control	JDK Mission Control (JMC) is a new JDK profiling and diagnostic tool platform for HotSpot JVM. It is a tool suite for basic monitoring, managing, and production time profiling and diagnostics with high performance. JMC minimizes the performance overhead that's usually an issue with profiling tools. See JDK Mission Control.
jcmd <b>utility</b>	This utility is used to send diagnostic command requests to the JVM, where these requests are useful for controlling Java Flight Recordings (JFRs). The JFRs are used to troubleshoot and diagnose JVM and Java applications with flight recording events.
jinfo <b>utility</b>	This utility can dynamically set, unset, and change the values of certain JVM flags for a specified Java process. On Oracle Solaris and Linux operating systems, it can also print configuration information.
jrunscript <b>utility</b>	This utility is a command-line script shell, which supports both interactive and batch-mode script execution.
Oracle Solaris Studio dbx debugger	This is an interactive, command-line debugging tool, which allows you to have complete control of the dynamic execution of a program, including stopping the program and inspecting its state. For details, see the latest dbx documentation located at Oracle Solaris Studio Program Debugging.
Oracle Solaris Studio Performance Analyzer	This tool can help you assess the performance of your code, identify potential performance problems, and locate the part of the code where the problems occur. The Performance Analyzer can be used from the command line or from a graphical user interface. For details, see the Oracle Solaris Studio Performance Analyzer.
Sun's Dataspace Profiling: DProfile	This tool provides insight into the flow of data within Sun computing systems, helping you identify bottlenecks in both software and hardware. DProfile is supported in the Sun Studio 11 compiler suite through the Performance Analyzer GUI. See DTrace or Dynamic Tracing diagnostic tool.
-Xcheck:jni option	This option is useful in diagnosing problems with applications that use the Java Native Interface (JNI) or that employ third-party libraries (some JDBC drivers, for example). See The -Xcheck:jni Option.
-verbose:class option	This option enables logging of class loading and unloading. See The - verbose:class Option.
-verbose:gc option	This option enables logging of garbage collection information. See The -verbose:gc Option.

#### Table 2-6 General Troubleshooting Tools and Options



Tool or Option	Description and Usage
-verbose:jni option	This option enables logging of JNI. See The -verbose:jni Option.
JAVA_TOOL_OPTIONS environment variable	This environment variable allows you to specify the initialization of tools, specifically the launching of native or Java programming language agents using the -agentlib or -javaagent options. See Environment Variables and System Properties.
java.security.debug system property	This system property controls whether the security checks in the JRE of the Java print trace messages during execution. See The java.security.debug System Property.

Table 2-6 (Cont.) General Troubleshooting Tools and O
---

# The java.lang.management Package

The java.lang.management package provides the management interface for the monitoring and management of the JVM and the operating system.

Specifically, it covers interfaces for the following systems:

- Class loading
- Compilation
- Garbage collection
- Memory manager
- Runtime
- Threads

In addition to the java.lang.management package, the JDK release includes platform extensions in the com.sun.management package. The platform extensions include a management interface to get detailed statistics from garbage collectors that perform collections in cycles. These extensions also include a management interface to get additional memory statistics from the operating system.

## The java.lang.instrument Package

The java.lang.instrument package provides services that allow the Java programming language agents to instrument programs running on the JVM.

Instrumentation is used by tools such as profilers, tools for tracing method calls, and many others. The package facilitates both load-time and dynamic instrumentation. It also includes methods to get information about the loaded classes and information about the amount of storage consumed by a given object.

## The java.lang.Thread Class

The java.lang.Thread class has a static method called getAllStackTraces, which returns a map of stack traces for all live threads.



The Thread class also has a method called getState, which returns the thread state; states are defined by the java.lang.Thread.State enumeration. These methods can be useful when you add diagnostic or monitoring capabilities to an application.

### JVM Tool Interface

The JVM Tool Interface (JVM TI) is a native (C/C++) programming interface that can be used by a wide range of development and monitoring tools.

JVM TI provides an interface for the full breadth of tools that need access to the VM state, including but not limited to profiling, debugging, monitoring, thread analysis, and coverage analysis tools.

Some examples of agents that rely on JVM TI are the following:

- Java Debug Wire Protocol (JDWP)
- The java.lang.instrument package

The specification for JVM TI can be found in the JVM Tool Interface documentation.

## The jrunscript Utility

The jrunscript utility is a command-line script shell.

It supports script execution in both interactive mode and in batch mode. By default, the shell uses JavaScript, but you can specify any other scripting language for which you supply the path to the script engine JAR file of .class files.

Thanks to the communication between the Java language and the scripting language, the jrunscript utility supports an exploratory programming style.

### The jstatd Daemon

The jstatd daemon is an RMI server application that monitors the creation and termination of each instrumented Java HotSpot, and provides an interface to allow remote monitoring tools to attach to JVMs running on the local host.

For example, this daemon allows the jps utility to list processes on a remote system.

#### Note:

The instrumentation is not accessible on FAT32 file system.

## Troubleshooting Tools Based on the Operating System

List of native Windows tools that can be used for troubleshooting problems.

Table 2-7 lists the troubleshooting tools available on the Windows operating system.

ΤοοΙ	Description
dumpchk	Command-line utility to verify that a memory dump file was created correctly. This tool is included in the Debugging Tools for Windows download available from the Microsoft website. See Collect Crash Dumps on Windows.
msdev <b>debugger</b>	Command-line utility that can be used to launch Visual C++ and the Win32 debugger
userdump	The User Mode Process Dumper is included in the OEM Support Tools download available from the Microsoft website. See Collect Crash Dumps on Windows.
windbg	Windows debugger can be used to debug Windows applications or crash dumps. This tool is included in the Debugging Tools for Windows download available from the Microsoft website. See Collect Crash Dumps on Windows.
/Md and /Mdd compiler options	Compiler options that automatically include extra support for tracking memory allocations

#### Table 2-7 Native Troubleshooting Tools on Windows

 Table 2-8 describes some troubleshooting tools introduced or improved in the Linux operating system version 10.

Tool	Description
c++filt	Demangle C++ mangled symbol names. This utility is delivered with the native C++ compiler suite: gcc on Linux.
gdb	GNU debugger
libnjamd	Memory allocation tracking
lsstack	Print thread stack (similar to pstack in the Oracle Solaris operating system)
	Not all distributions provide this tool by default; therefore, you might have to download it from Open Source downloads.
ltrace	Library call tracer (equivalent to truss -u in the Oracle Solaris operating system)
	Not all distributions provide this tool by default; therefore, you might have to download it from Open Source downloads.
mtrace <b>and</b> muntrace	GNU malloc tracer
proc <b>tools such as</b> pmap <b>and</b> pstack	Some, but not all, of the proc tools on the Oracle Solaris operating system have equivalent tools on Linux. Core file support is not as good for Linux as for Oracle Solaris operating system; for example, pstack does not work for core dumps
strace	System call tracer (equivalent to truss -t in the Oracle Solaris operating system)
top	Display most CPU-intensive processes.
vmstat	Report information about processes, memory, paging, block I/O, traps, and CPU activity.

#### Table 2-8 Native Troubleshooting Tools on Linux

 Table 2-9 lists troubleshooting tools available on Oracle Solaris operating system.

ORACLE

Tool	Description
coreadm	Specify name and location of core files produced by the JVM.
cpustat	Monitor system behavior using CPU performance counters.
cputrack	Monitor process and LWP behavior using CPU performance counters.
c++filt	Demangle C++ mangled symbol names. This utility is delivered with the native C++ compiler suite: SUNWspro on the Oracle Solaris operating system.
dtrace	Introduced in Oracle Solaris 10 operating system, DTrace is a dynamic tracing compiler and tracing utility. It can perform dynamic tracing of kernel functions, system calls, and user functions. This tool allows arbitrary, safe scripting to be executed at entry, exit, and other probe points. The script is written in the C-like, but safe, pointer semantics language called the D programming language. See also DTrace Tool.
gcore	Force a core dump of a process. The process continues after the core dump is written.
intrstat	Report statistics on the CPU consumed by interrupt threads.
iostat	Report I/O statistics.
libumem	Introduced in the Oracle Solaris 9 operating system update 3, this library provides fast, scalable object-caching memory allocation and extensive debugging support. The tool can be used to find and fix memory management bugs. See Find Leaks with the libumem Tool.
mdb	Modular debugger for kernel and user applications and crash dumps
netstat	Display the contents of various network-related data structures.
pargs	Print process arguments, environment variables, or the auxiliary vector. Long output is not truncated as it would be by other commands, such as ps.
pfiles	Print information on process file descriptors. Starting with the Oracle Solaris 10 operating system, the tool prints the file name also.
pldd	Print shared objects loaded by a process.
pmap	Print memory layout of a process or core file, including heap, data, and text sections. Starting with Oracle Solaris 10, stack segments are clearly identified with the text [stack] along with the thread ID. See Improvements to the pmap Utility.
prstat	Report statistics for active Oracle Solaris operating system processes. (Similar to top)
prun	Set the process to running mode (reverse of pstop).
ps	List all processes.
psig	List the signal handlers of a process.
pstack	Print stack of threads of a given process or core file. Starting with the Oracle Solaris 10 operating system, Java method names can be printed for Java frames. See Improvements to the pstack Utility.
pstop	Stop the process (suspend).
ptree	Print the process tree that contains the given PID.
sar	System activity reporter

#### Table 2-9 Native Troubleshooting Tools on Oracle Solaris Operating System



Tool	Description
sdtprocess	Display most CPU-intensive processes. (similar to top).
sdtperfmeter	Display graphs that show the system performance (for example, CPU, disks, and network).
top	Display most CPU-intensive processes. This tool is available as freeware for the Oracle Solaris operating system, but is not installed by default.
trapstat	Display runtime trap statistics (SPARC only).
truss	Trace entry and exit events for system calls, user-mode functions, and signals; optionally stop the process at one of these events. This tool also prints the arguments of system calls and user functions.
vmstat	Report system virtual memory statistics.
watchmalloc	Track memory allocations.

#### Table 2-9 (Cont.) Native Troubleshooting Tools on Oracle Solaris Operating System

# 3 Troubleshoot Memory Leaks

This chapter provides some suggestions for diagnosing problems involving possible memory leaks.

If your application's execution time becomes longer, or if the operating system seems to be performing slower, this could be an indication of a memory leak. In other words, virtual memory is being allocated but is not being returned when it is no longer needed. Eventually the application or the system runs out of memory, and the application terminates abnormally.

This chapter contains the following sections:

- Use JDK Mission Control to Debug Memory Leak
- The jfr tool
- Understand the OutOfMemoryError Exception
- Troubleshoot a Crash Instead of OutOfMemoryError
- Diagnose Leaks in Java Language Code
- Diagnose Leaks in Native Code

## Use JDK Mission Control to Debug Memory Leak

The Flight Recorder records detailed information about the Java runtime and the Java applications running in the Java runtime.

The following sections describe how to debug a memory leak by analyzing a flight recording in JMC.

- Detect Memory Leak
- Find the Leaking Class

#### Detect Memory Leak

You can detect memory leaks early and prevent OutOfmemoryErrors using JMC.

Detecting a slow memory leak can be hard. A typical symptom could be the application becoming slower after running for a long time due to frequent garbage collections. Eventually, OutOfmemoryErrors may be seen. However, memory leaks can be detected early, even before such problems occur, by analyzing Java Flight recordings.

Watch if the live set of your application is increasing over time. The live set is the amount of Java heap that is used after an old collection (all objects that are not live) and have been garbage collected. To inspect the live set, open JMC and connect to a JVM using the Java Management console (JMX). Open the **MBean Browser** tab and look for the GarbageCollectorAggregator MBean under com.sun.management.

Open JMC and start a Time fixed recording (profiling recording) for an hour. Before starting a flight recording, make sure that the option **Object Types + Allocation Stack Traces + Path to GC Root** is selected from the **Memory Leak Detection** setting.



Once the recording is complete, the recording file (.jfr) opens in JMC. Look at the **Automated Analysis Results** page. To detect a memory leak focus on the **Live Objects** section of the page. Here is a sample figure of a recording, which shows heap size issue:

JDK Mission Control Eile Edit Navigate Window Help		- 0 ×
🗑 JVM Browser 📴 Outline 🛛 👘 💝 🐃 🗖	🛥 Leak jfr 🕸	0
Mutomated Analysis Results	Automated Analysis Results	<b>e</b> (a) <b>e</b>
<ul> <li>Zava Application</li> <li>Threads</li> <li>* Memory</li> </ul>	Java Application	
tive Objects	33 Threads Allocating	+ Stackdepth Setting +
	42 Parallel Threads	Garbage Collections
+@ Socket I/O	<ul> <li>Memory</li> </ul>	100 G1/CMS Full Collection -
<ul> <li>Method Profiling</li> <li>Exceptions</li> <li>Thread Dumps</li> </ul>	42     Allocated Classes     4       88     Heap Content     4	At least one Full, Stop-The-World Garbage Collection occurred during this recording. For the CMS and G1 collectors, Full GC events are a strong negative performance indicator. Tunable GC parameters can be used to allow the collector to operate in concurrent mode, avoiding Stop-The-World pauses and increasing
<ul> <li></li></ul>	Most of the heap was used by only a few classes. If the heap usage needs to be reduced, then this would be a good place to start.	GC and application performance.
<ul> <li>         • Q. Compilations         • Class Loading         # VM Operations     </li> </ul>	Heap Live Set Trend The live set on the heap seems to increase with a speed of about 12.2 KiB per second during the recording. An analysis of the reference tree found 1 leak candidates.	<ul> <li>Inspection garbage collections may be a problem since they usually take a lot of time.</li> <li>Environment</li> </ul>
TLAB Allocations	<ul> <li>Java.util.Hashtable\$Entry[6534/98303]</li> <li>The candidate is referenced by this chain.</li> </ul>	S Processes
G Environment     G Processes	java.util.Hashtable.table (Stack Variable : Threads (Thread	Competing CPU Ratio Usage -
Properties  Results	Name: main))	An average CPU load of 45 % was caused by other processes during 32.415 s starting at 7/7/20, 5:05:24 PM. The apolication performance can be affected when the machine is under heavy
Field Value 0 events		load and there are other processes that use CPU or other resources on the same

Figure 3-1 Memory Leak - Automated Analysis Page

You can observe that in the **Heap Live Set Trend** section, the live set on the heap seems to increase rapidly and the analysis of the reference tree detected a leak candidate.

For further analysis, open the **Java Applications** page and then click the **Memory** page. Here is a sample figure of a recording, which shows memory leak issue.

Figure 3-2	Memory	/ Leak -	Memory	/ Page
------------	--------	----------	--------	--------

JDK Mission Control Eile Edit Navigate Window Help								- 0	×
JVM Browser & Outline	∞ flight recording 1401Leak31092.ifr ∺								** E
Automated Analysis Results     Automated Analysis     Automated Analysis Results     Automated Analysis Results     Au	Memory <pre></pre>	<b>x</b> î	~		Show co	ncurrent: Conti	ained 🗹 Same thread	ds Time Range: Se	0   = et Clear
<ul> <li>Memory</li> </ul>			3115						
Live Objects Lock Instances	Class	Max Live Co	Max Live Size	Live Size Inc	Total Allocat				^
d <sup>a</sup> LOCK Instances +8 File I/O	<sup>(9</sup> char[]	51,723	26.1 MiB	25.5 MiB	10.2 GiB				
+8 File I/O +8 Socket I/O	<sup>☉</sup> java.util.TreeMap\$Entry	107,783	4.11 MiB	1.13 MiB	171 MiB				
Method Profiling	<sup>©</sup> Leak\$DemoObject	51,655	1.97 MiB	1.93 MiB	505 MiB				
* Exceptions	<sup>©</sup> java.util.Hashtable\$Entry	51,663	1.58 MiB	1.54 MiB	252 MiB				
P Thread Dumps	<sup>©</sup> java.util.TreeMap	34,317	1.57 MiB	441 KiB	97.6 MiB				
	<sup>©</sup> java.lang.Long	54,521	1.25 MiB	354 KiB	3.75 MiB				
Properties ** Results	<sup>©</sup> byte[]	21,761	1.18 MiB	121 KiB	57.7 MiB				
88 Heap Content - 🔨	☺ iavax.management.openmbean.CompositeData	Support 34.277	803 KiB	221 KiB	49.7 MiB				~
Most of the heap was used by only a few classes. If the heap usage needs to be reduced, then this would be a good place to start.	768 MIB + 512 MIB + 256 MIB - 256 MIB -							Garbage Col	ion
Allocated Classes -					· · · · · · · · · · · · · · · · · · ·			Heap Space	
The most allocated class is likely char[]'. This is the most common allocation path for that class:	64 MiB 32 MiB	with the second seco						Heap Space	: Rest
Leak\$DemoObject. <init>(int) (96.9 %) Frequently allocated classes are likely good places to start when trying to</init>	7/7/2020	4:30:00 PM	4	:45:00 PM	5:0	10:00 PM	5:15:00 P	<	<b>`</b>
reduce garbage collections. Look at the aggregated stack traces of the	© Progress View     ≡ Stack Trace							K % K   ≋   <b>⊙</b> ≯	~ ~ 0
nost commonly allocated classes to see if many instances are created	Stack Trace			Count					
along the same call path. Try to	tvoid Leak\$DemoObject. <init>(int)</init>			198499					
educe the number of instances	\$ void Leak\$DemoThread.put(int)			100441					
created by invoking the most commonly taken paths less.	t void Leak\$DemoThread.run()			100441					
C Pressure +	1 void java.lang.Thread.run()			100441					
12 String Deduplication +									

You can observe from the graph that the memory usage has increased steadily, which indicates a memory leak issue.



## Find the Leaking Class

You can use the Java Flight Recordings to identify the leaking class.

To find the leaking class, open the **Memory** page and click the **Live Objects** page. Here is a sample figure of a recording, which shows the leaking class.

Figure 3-3 Memory Leak - Live Objects Page

🛿 Automal 😸 🗸 🗸	ted Analysis Results	🚓 Live Objects						0
<ul> <li>Java App</li> <li>Threa</li> <li>Mem</li> </ul>	ds	Allocation	1- <b>F</b>	1		• <b>F</b> •		
🕫 Liv	ve Objects	7/7/2020	4:20:00 PM	4:30:00 PM	4:40:00 PM	4:50:00 PM	5:00:00 PM	5:10:00 PM
₽ Lock S File I/	Instances O	Live Object Sample	×		Count De	scription		
- Socke	et I/O	✓ Hashtable			173 Sta	ck Variable : Threads (Thread Na		
O Meth	od Profiling	✓ Hashtable\$Entry[]			173			
1 Excep	tions	✓ Hashtable\$Entry	,		1			
Threa	d Dumps v	✓ Leak\$DemoC	Dbject		1			
Properties	Results 📑 🗢 🗆	char[] > Hashtable\$Entry			1			
Field	Value	> Hashtable\$Entry		-	1			
Event Type	Old Object Sample	> Hashtable\$Entry			1			
	7/7/20, 5:15:49 PM	> Hashtable\$Entry		-				
Duration	91.053 ms	> Hashtable\$Entry			1			
SEnd Time	7/7/20, 5:15:49 PM	> Hashtable\$Entry			1			
Event Thre	Thread-1	> Hashtable\$Entry	,		1			
Allocation	7/7/20, 4:13:15 PM	> Hashtable\$Entry	,		1			
Object Age	1 h 3 min	> Hashtable\$Entry	,		1			
Last Know	4.56 MiB	> Hashtable\$Entry	,		1			
■ Object	char[].chunk @ 0x70	> Hashtable\$Entry	1		1			
Array Ele	255							
GC Root	Stack Variable : Threa	Progress View Stack T	race					K N K   €   <b>O</b> №
	1 events	Stack Trace			Count			
		1 void Leak\$DemoObject.	<init>(int)</init>		1			
		1 void Leak\$DemoThread.	.put(int)		1			
		t void Leak\$DemoThread	.run()		1			
		1 void java.lang.Thread.ru	n()		1			

You can observe that most of the live objects being tracked are actually held on to by Leak\$DemoThread, which in turn holds on to a leaked char[] class. For further analysis, see the Old Object Sample event in the **Results** tab that contains sampling of the objects that have survived. This event contains the time of allocation, the allocation stack trace, the path back to the GC root.

When a potentially leaking class is identified, look at the **TLAB Allocations** page in the **JVM Internals** page for some samples of where objects were allocated. Here is a sample figure of a recording, which shows TLAB allocations.



1 Exceptions	TLAB Allocations						
记 JVM Internals	<no selection=""></no>	~		Show concu	rrent: 🗹 Conta	ined 🗹 Same thr	eads Time Range: Set Clear
Garbage Collections GC Configuration	Thread	Cour	t Average TL	Average All	Ect TLAR AL	Total Allocat	
> 9. Compilations	*Transient Allocator	124,17		4.08 KiB	14.6 GiB	58.9 MiB	
Class Loading	RMI TCP Connection(idle)	37.40		4.08 KIB 3.31 KIB	370 MiB	31.1 MiB	
& VM Operations	RMI TCP Connection(idle)	29.23		3.31 KIB 3.97 KiB	250 MiB	26.2 MiB	
TLAB Allocations						20.2 MIB 15.9 MIB	
Environment	RMI TCP Connection(idle)	12,49		6.45 KiB	281 MiB		
Processes	RMI TCP Connection(idle)	10,82		5.6 KiB	132 MiB	12.5 MiB	
Properties and Results	RMI TCP Connection(idle)	15,02		2.44 KiB	162 MiB	10.4 MiB	
	RMI TCP Connection(idle)	12,88		3.52 KiB	133 MiB	10.2 MiB	
1 TLAB Allocation Ratio +	RMI TCP Connection(idle)	7,47		6.53 KiB	155 MiB	10 MiB	
	RMI TCP Connection(idle)	7,00		6.25 KiB	135 MiB	8.48 MiB	
	RMI TCP Connection(idle)	6,46		6.27 KiB	154 MiB	8.11 MiB	
	RMI TCP Connection(idle)	5,94		6.47 KiB	103 MiB	7.24 MiB	
	RMI TCP Connection(idle)	11,85		1.87 KiB	130 MiB	6.2 MiB	
	RMI TCP Connection(idle)	3,80		5.49 KiB	49.6 MiB	4.21 MiB	
	RMI TCP Connection(idle)	6,30	82.9 B	1.91 KiB	55.3 MiB	3.28 MiB	
	RMI TCP Connection(idle)	92	7 113 B	7.93 KiB	17.1 MiB	1.3 MiB	
	≫ Thread-0	106,69	7 494 B	1.6 KiB	5.44 GiB	707 KiB	
	1.25 GB 7 1 GB 7 758 MB 7 512 MB 7 255 MB 7 748 Allocations (1 thread)						Est. TLAB Allocation Out
	7/7/2020	4:30:00 PM 4:45:0		5:00:0		5:15:00	

Figure 3-4 Memory Leak - TLAB Allocations

Check the class samples being allocated. If the leak is slow, there may be a few allocations of this object and may be no samples. Also, it may be that only a specific allocation site is leading to a leak. You can make required changes to the code to fix the leaking class.

## The jfr tool

Java Flight Recorder (JFR) records detailed information about the Java runtime and the Java application running on the Java runtime. This information can be used to identify memory leaks.

To detect a memory leak, JFR must be running at the time that the leak occurs. The overhead of JFR is very low, less than 1%, and it has been designed to be safe to have always on in production.

Start a recording when the application is started using the java command as shown in the following example:

```
$ java -XX:StartFlightRecording
```

When the JVM runs out of memory and exits due to a <code>java.lang.OutOfMemoryError</code> error, a recording with the prefix <code>hs\_oom\_pid</code> is often, but not always, written to the directory in which the JVM was started. An alternative way to get a recording is to dump it before the application runs out of memory using the <code>jcmd</code> tool, as shown in the following example:

```
$ jcmd pid JFR.dump filename=recording.jfr path-to-gc-roots=true
```

When you have a recording, use the jfr tool located in the *java-home/bin* directory to print Old Object Sample events that contain information about potential memory



leaks. The following example shows the command and an example of the output from a recording for an application with the pid 16276:

```
jfr print --events OldObjectSample pid16276.jfr
. . .
jdk.OldObjectSample {
  startTime = 18:32:52.192
 duration = 5.317 s
  allocationTime = 18:31:38.213
  objectAge = 74.0 s
  lastKnownHeapUsage = 63.9 MB
 object = [
    java.util.HashMap$Node
    [15052855] : java.util.HashMap$Node[33554432]
    table : java.util.HashMap Size: 15000000
   map : java.util.HashSet
   users : java.lang.Class Class Name: Application
  1
  arrayElements = N/A
  root = \{
   description = "Thread Name: main"
    system = "Threads"
   type = "Stack Variable"
  }
  eventThread = "main" (javaThreadId = 1)
}
. . .
jdk.OldObjectSample {
 startTime = 18:32:52.192
  duration = 5.317 s
  allocationTime = 18:31:38.266
  objectAge = 74.0 s
 lastKnownHeapUsage = 84.4 MB
  object = [
    java.util.HashMap$Node
    [8776975] : java.util.HashMap$Node[33554432]
   table : java.util.HashMap Size: 15000000
   map : java.util.HashSet
   users : java.lang.Class Class Name: Application
  ]
  arrayElements = N/A
  root = \{
   description = "Thread Name: main"
    system = "Threads"
    type = "Stack Variable"
  }
 eventThread = "main" (javaThreadId = 1)
}
. . .
jdk.OldObjectSample {
```



```
startTime = 18:32:52.192
 duration = 5.317 s
 allocationTime = 18:31:38.540
 objectAge = 73.7 s
 lastKnownHeapUsage = 121.7 MB
 object = [
   java.util.HashMap$Node
   [393162] : java.util.HashMap$Node[33554432]
   table : java.util.HashMap Size: 15000000
   map : java.util.HashSet
   users : java.lang.Class Class Name: Application
 1
 arrayElements = N/A
 root = \{
   description = "Thread Name: main"
   system = "Threads"
   type = "Stack Variable"
 }
 eventThread = "main" (javaThreadId = 1)
. . .
```

To identify a possible memory leak, review the following elements in the recording:

- First, notice that the lastKnownHeapUsage element in the Old Object Sample events is increasing over time, from 63.9 MB in the first event in the example to 121.7 MB in the last event. This increase is an indication that there is a memory leak. Most applications allocate objects during startup and then allocate temporary objects that are periodically garbage collected. Objects that are not garbage collected, for whatever reason, accumulate over time and increase the value of lastKnownHeapUsage.
- Next, look at the allocationTime element to see when the object was allocated. Objects that are allocated during startup are typically not memory leaks, neither are objects allocated close to when the dump was taken. The objectAge element shows how long the object has been alive. The startTime and duration elements are not related to when the memory leak occurred, but when the OldObject event was emitted and how long it took to gather data for it. This information can be ignored.
- Then look at the object element to see the memory leak candidate; in this example, an object of type java.util.HashMap\$Node. It is held by the table field in the java.util.HashMap class, which is held by java.util.HashSet, which in turn is held by the users field of the Application class.
- The root element contains information about the GC root. In this example, the Application class is held by a stack variable in the main thread. The eventThread element provides information about the thread that allocated the object.

}

If the application is started with the -XX:StartFlightRecording:settings=profile option, then the recording also contains the stack trace from where the object was allocated, as shown in the following example:

```
stackTrace = [
   java.util.HashMap.newNode(int, Object, Object, HashMap$Node) line: 1885
   java.util.HashMap.putVal(int, Object, Object, boolean, boolean) line: 631
   java.util.HashMap.put(Object, Object) line: 612
   java.util.HashSet.add(Object) line: 220
   Application.storeUser(String, String) line: 53
   Application.validate(String, String) line: 48
   Application.login(String, String) line: 44
   Application.main(String[]) line: 30
}
```

In this example we can see that the object was put in the HashSet when the storeUser(String, String) method was called. This suggests that the cause of the memory leak might be objects that were not removed from the HashSet when the user logged out.

It is not recommended to always run all applications with the – XX:StartFlightRecording:settings=profile option due to overhead in certain allocationintensive applications, but is typically OK when debugging. Overhead is usually less than 2%.

Setting <code>path-to-gc-roots=true</code> creates overhead, similar to a full garbage collection, but also provides reference chains back to the GC root, which is usually sufficient information to find the cause of a memory leak.

## Understand the OutOfMemoryError Exception

java.lang.OutOfMemoryError error is thrown when there is insufficient space to allocate an object in the Java heap.

One common indication of a memory leak is the java.lang.OutOfMemoryError exception. In this case, the garbage collector cannot make space available to accommodate a new object, and the heap cannot be expanded further. Also, this error may be thrown when there is insufficient native memory to support the loading of a Java class. In a rare instance, a java.lang.OutOfMemoryError can be thrown when an excessive amount of time is being spent doing garbage collection, and little memory is being freed.

When a java.lang.OutOfMemoryError exception is thrown, a stack trace is also printed.

The java.lang.OutOfMemoryError exception can also be thrown by native library code when a native allocation cannot be satisfied (for example, if swap space is low).

An early step to diagnose an OutOfMemoryError exception is to determine the cause of the exception. Was it thrown because the Java heap is full, or because the native heap is full? To help you find the cause, the text of the exception includes a detail message at the end, as shown in the following exceptions:

Exception in thread thread\_name: java.lang.OutOfMemoryError: Java heap space Cause: The detailed message Java heap space indicates that an object could not be allocated in the Java heap. This error does not necessarily imply a memory leak. The



problem can be as simple as a configuration issue, where the specified heap size (or the default size, if it is not specified) is insufficient for the application. In other cases, and in particular for a long-lived application, the message might be an indication that the application is unintentionally holding references to objects, and this prevents the objects from being garbage collected. This is the Java language equivalent of a memory leak.

#### Note:

The APIs that are called by an application could also be unintentionally holding object references.

One other potential source of this error arises with applications that make excessive use of finalizers. If a class has a finalize method, then objects of that type do not have their space reclaimed at garbage collection time. Instead, after garbage collection, the objects are queued for finalization, which occurs at a later time. In the Oracle Sun implementation, finalizers are executed by a daemon thread that services the finalization queue. If the finalizer thread cannot keep up with the finalization queue, then the Java heap could fill up, and this type of OutOfMemoryError exception would be thrown. One scenario that can cause this situation is when an application creates high-priority threads that cause the finalization queue to increase at a rate that is faster than the rate at which the finalizer thread is servicing that queue.

Action: Try increasing the Java heap size. See Monitor the Objects Pending Finalization to learn more about how to monitor objects for which finalization is pending. See Finalization and Weak, Soft, and Phantom References in Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide for information about detecting and migrating from finalization.

## Exception in thread *thread\_name*: java.lang.OutOfMemoryError: GC Overhead limit exceeded

Cause: The detail message "GC overhead limit exceeded" indicates that the garbage collector is running all the time, and the Java program is making very slow progress. After a garbage collection, if the Java process is spending more than approximately 98% of its time doing garbage collection and if it is recovering less than 2% of the heap and has been doing so for the last 5 (compile time constant) consecutive garbage collections, then a java.lang.OutOfMemoryError is thrown. This exception is typically thrown because the amount of live data barely fits into the Java heap having little free space for new allocations.

Action: Increase the heap size. The java.lang.OutOfMemoryError exception for GC Overhead limit exceeded can be turned off with the command-line flag -xx:-UseGCOverheadLimit.

**Exception in thread** *thread\_name*: Requested array size exceeds VM limit Cause: The detail message "Requested array size exceeds VM limit" indicates that the application (or APIs used by that application) attempted to allocate an array with a size larger than the VM implementation limit, irrespective of how much heap size is available.

Action: Ensure that your application (or APIs used by that application) allocates an array with a size less than the VM implementation limit.



#### Exception in thread thread\_name: java.lang.OutOfMemoryError: Metaspace

Cause: Java class metadata (the virtual machines internal presentation of Java class) is allocated in native memory (referred to here as metaspace). If metaspace for class metadata is exhausted, a java.lang.OutOfMemoryError exception with a detail MetaSpace is thrown. The amount of metaspace that can be used for class metadata is limited by the parameter MaxMetaSpaceSize, which is specified on the command line. When the amount of native memory needed for a class metadata exceeds MaxMetaSpaceSize, a java.lang.OutOfMemoryError exception with a detail MetaSpace is thrown.

Action: If MaxMetaSpaceSize, has been set on the command-line, increase its value. MetaSpace is allocated from the same address spaces as the Java heap. Reducing the size of the Java heap will make more space available for MetaSpace. This is only a correct tradeoff if there is an excess of free space in the Java heap. See the following action for **Out of swap space** detailed message.

## Exception in thread *thread\_name*: java.lang.OutOfMemoryError: request *size* bytes for *reason*. Out of swap space?

Cause: The detail message "request *size* bytes for *reason*. Out of swap space?" appears to be an OutOfMemoryError exception. However, the Java HotSpot VM code reports this apparent exception when an allocation from the native heap failed and the native heap might be close to exhaustion. The message indicates the size (in bytes) of the request that failed and the reason for the memory request. Usually the reason is the name of the source module reporting the allocation failure, although sometimes it is the actual reason.

Action: When this error message is thrown, the VM invokes the fatal error handling mechanism (that is, it generates a fatal error log file, which contains useful information about the thread, process, and system at the time of the crash). In the case of native heap exhaustion, the heap memory and memory map information in the log can be useful. See Fatal Error Log.

If this type of the OutOfMemoryError exception is thrown, you might need to use troubleshooting utilities on the operating system to diagnose the issue further. See Native Operating System Tools.

## Exception in thread *thread\_name*: java.lang.OutOfMemoryError: Compressed class space

Cause: On 64-bit platforms, a pointer to class metadata can be represented by 32-bit offset (with UseCompressedOops). This is controlled by the command line flag

UseCompressedClassPointers (on by default). If the UseCompressedClassPointers is used, the amount of space available for class metadata is fixed at the amount

CompressedClassSpaceSize. If the space needed for UseCompressedClassPointers exceeds CompressedClassSpaceSize, a java.lang.OutOfMemoryError with detail Compressed class space is thrown.

Action: Increase CompressedClassSpaceSize to turn off UseCompressedClassPointers. Note: There are bounds on the acceptable size of CompressedClassSpaceSize. For example -XX: CompressedClassSpaceSize=4g, exceeds acceptable bounds will result in a message such as

CompressedClassSpaceSize of 4294967296 is invalid; must be between 1048576 and 3221225472.



#### Note:

There is more than one kind of class metadata, -klass metadata, and other metadata. Only klass metadata is stored in the space bounded by CompressedClassSpaceSize. The other metadata is stored in Metaspace.

## Exception in thread *thread\_name*: java.lang.OutOfMemoryError: *reason* stack\_trace\_with\_native\_method

Cause: If the detail part of the error message is "reason

*stack\_trace\_with\_native\_method*, and a stack trace is printed in which the top frame is a native method, then this is an indication that a native method, has encountered an allocation failure. The difference between this and the previous message is that the allocation failure was detected in a Java Native Interface (JNI) or native method rather than in the JVM code.

Action: If this type of the OutOfMemoryError exception is thrown, you might need to use native utilities of the OS to further diagnose the issue. See Native Operating System Tools.

## Troubleshoot a Crash Instead of OutOfMemoryError

Use the information in the fatal error log or the crash dump to troubleshoot a crash.

Sometimes an application crashes soon after an allocation from the native heap fails. This occurs with native code that does not check for errors returned by the memory allocation functions.

For example, the malloc system call returns null if there is no memory available. If the return from malloc is not checked, then the application might crash when it attempts to access an invalid memory location. Depending on the circumstances, this type of issue can be difficult to locate.

However, sometimes the information from the fatal error log or the crash dump is sufficient to diagnose this issue. The fatal error log is covered in detail in Fatal Error Log. If the cause of the crash is an allocation failure, then determine the reason for the allocation failure. As with any other native heap issue, the system might be configured with the insufficient amount of swap space, another process on the system might be consuming all memory resources, or there might be a leak in the application (or in the APIs that it calls) that causes the system to run out of memory.

## Diagnose Leaks in Java Language Code

Use the NetBeans profiler to diagnose leaks in the Java language code.

Diagnosing leaks in the Java language code can be difficult. Usually, it requires very detailed knowledge of the application. In addition, the process is often iterative and lengthy. This section provides information about the tools that you can use to diagnose memory leaks in the Java language code.



#### Note:

Beside the tools mentioned in this section, a large number of third-party memory debugger tools are available. The Eclipse Memory Analyzer and YourKit are two examples of commercial tools with memory debugging capabilities. There are many others, and no specific product is recommended.

The following utilities used to diagnose leaks in the Java language code.

The NetBeans Profiler: The NetBeans Profiler can locate memory leaks very quickly. Commercial memory leak debugging tools can take a long time to locate a leak in a large application. The NetBeans Profiler, however, uses the pattern of memory allocations and reclamations that such objects typically demonstrate. This process includes also the lack of memory reclamations. The profiler can check where these objects were allocated, which often is sufficient to identify the root cause of the leak.

See Introduction to Profiling Java Applications in NetBeans IDE.

The following sections describe the other ways to diagnose leaks in the Java language code.

- Get a Heap Histogram
- Monitor the Objects Pending Finalization

### Get a Heap Histogram

Get a heap histogram to identify memory leaks using the different commands and options available.

You can try to quickly narrow down a memory leak by examining the heap histogram. You can get a heap histogram in several ways:

- If the Java process is started with the -XX:+PrintClassHistogram command-line option, then the Control+Break handler will produce a heap histogram.
- You can use the jmap utility to get a heap histogram from a running process:

It is recommended to use the latest utility, jcmd, instead of jmap utility for enhanced diagnostics and reduced performance overhead. See Useful Commands for the jcmd Utility. The command in the following example creates a heap histogram for a running process using jcmd and results similar to the following jmap command.

jcmd <process id/main class> GC.class histogram filename=Myheaphistogram

jmap -histo pid

The output shows the total size and instance count for each class type in the heap. If a sequence of histograms is obtained (for example, every 2 minutes), then you might be able to see a trend that can lead to further analysis.



• You can use the jhsdb jmap utility to get a heap histogram from a core file, as shown in the following example.

jhsdb jmap --histo --exe jdk-home/bin/java --corecore file

For example, if you specify the -XX:+CrashOnOutOfMemoryError commandline option while running your application, then when an OutOfMemoryError exception is thrown, the JVM will generate a core dump. You can then execute jmap on the core file to get a histogram, as shown in the following example.

```
$ jhsdb jmap --histo --exe /usr/java/jdk-11/bin/java --core
core.21844
```

```
$ Attaching to core core.21844 from executable /usr/java/jdk-11/bin/
java, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 11-ea+24
Iterating over heap. This may take a while...
Object Histogram:
```

```
#instances #bytes Class description
nıım
_____
_____
1:
              2108 112576 byte[]

      546
      66112
      java.lang.Class

      1771
      56672
      java.util.HashMap$Node

      574
      53288
      java.lang.Object[]

      1860
      44640
      java.lang.String

2:
3:
4:
5:
       349 40016 java.util.HashMap$Node[]
16 33920 char[]
977 31264
6:
7:
8:
java.util.concurrent.ConcurrentHashMap$Node
        327 15696 java.util.HashMap
9:
               266
10:
                          13800 java.lang.String[]
11:
               485
                         12880 int[]
•
Total : 14253 633584
Heap traversal took 1.15 seconds.
```

The above example shows that the OutOfMemoryError exception was caused by the number of byte arrays (2108 instances in the heap). Without further analysis it is not clear where the byte arrays are allocated. However, the information is still useful.

## Monitor the Objects Pending Finalization

Different commands and options available to monitor the objects pending finalization.

When the OutOfMemoryError exception is thrown with the "Java heap space" detail message, the cause can be excessive use of finalizers. To diagnose this, you have several options for monitoring the number of objects that are pending finalization:



- The JConsole management tool can be used to monitor the number of objects that are pending finalization. This tool reports the pending finalization count in the memory statistics on the **Summary** tab pane. The count is approximate, but it can be used to characterize an application and understand if it relies a lot on finalization.
- On Oracle Solaris and Linux operating systems, the jmap utility can be used with the finalizerinfo option to print information about objects awaiting finalization.
- An application can report the approximate number of objects pending finalization using the getObjectPendingFinalizationCount method of the java.lang.management.MemoryMXBean class. Links to the API documentation and example code can be found in Custom Diagnostic Tools. The example code can easily be extended to include the reporting of the pending finalization count.

See Finalization and Weak, Soft, and Phantom References in *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide* for information about detecting and migrating from finalization.

## Diagnose Leaks in Native Code

Several techniques can be used to find and isolate native code memory leaks. In general, there is no ideal solution for all platforms.

The following are some techniques to diagnose leaks in native code.

- Track All Memory Allocation and Free Calls
- Track All Memory Allocations in the JNI Library
- Track Memory Allocation with Operating System Support
- Find Leaks with the dbx Debugger
- Find Leaks with the libumem Tool

## Track All Memory Allocation and Free Calls

Tools available to track all memory allocation and use of that memory.

A very common practice is to track all allocation and free calls of the native allocations. This can be a fairly simple process or a very sophisticated one. Many products over the years have been built up around the tracking of native heap allocations and the use of that memory.

Tools like IBM Rational Purify and the runtime checking functionality of Sun Studio dbx debugger can be used to find these leaks in normal native code situations and also find any access to native heap memory that represents assignments to un-initialized memory or accesses to freed memory. See Find Leaks with the dbx Debugger.

Not all these types of tools will work with Java applications that use native code, and usually these tools are platform-specific. Because the virtual machine dynamically creates code at runtime, these tools can incorrectly interpret the code and fail to run at all, or give false information. Check with your tool vendor to ensure that the version of the tool works with the version of the virtual machine you are using.

See sourceforge for many simple and portable native memory leak detecting examples. Most libraries and tools assume that you can recompile or edit the source of the application and place wrapper functions over the allocation functions. The more powerful of these tools allow you to run your application unchanged by interposing over these allocation functions



dynamically. This is the case with the library libumem.so first introduced in the Oracle Solaris 9 operating system update 3; see Find Leaks with the libumem Tool.

## Track All Memory Allocations in the JNI Library

If you write a JNI library, then consider creating a localized way to ensure that your library does not leak memory, by using a simple wrapper approach.

The procedure in the following example is an easy localized allocation tracking approach for a JNI library. First, define the following lines in all source files.

```
#include <stdlib.h>
#define malloc(n) debug_malloc(n, __FILE_, __LINE_)
#define free(p) debug_free(p, __FILE_, __LINE_)
```

Then, you can use the functions in the following example to watch for leaks.

```
/* Total bytes allocated */
static int total allocated;
/* Memory alignment is important */
typedef union { double d; struct {size t n; char *file; int line;}
s; } Site;
void *
debug malloc(size t n, char *file, int line)
    char *rp;
    rp = (char*)malloc(sizeof(Site)+n);
    total allocated += n;
    ((Site^*)rp) \rightarrow s.n = n;
    ((Site*)rp)->s.file = file;
    ((Site*)rp)->s.line = line;
    return (void*) (rp + sizeof(Site));
}
void
debug free(void *p, char *file, int line)
{
    char *rp;
    rp = ((char^*)p) - sizeof(Site);
    total allocated -= ((Site*)rp)->s.n;
    free(rp);
}
```

The JNI library would then need to periodically (or at shutdown) check the value of the total\_allocated variable to verify that it made sense. The preceding code could also be expanded to save in a linked list the allocations that remained, and report where the leaked memory was allocated. This is a localized and portable way to track memory allocations in a single set of sources. You would need to ensure that debug\_free() was called only with the pointer that came from debug\_malloc(), and you would also need to create similar functions for realloc(), calloc(), strdup(), and so forth, if they were used.

A more global way to look for native heap memory leaks involves interposition of the library calls for the entire process.



## Track Memory Allocation with Operating System Support

Tools available for tracking memory allocation in an operating system.

Most operating systems include some form of global allocation tracking support.

- On Windows, search the MSDN library for debug support. The Microsoft C++ compiler has the /Md and /Mdd compiler options that will automatically include extra support for tracking memory allocation.
- Linux systems have tools such as mtrace and libnjamd to help in dealing with allocation tracking.
- The Oracle Solaris operating system provides the watchmalloc tool. Oracle Solaris 9 operating system update 3 also introduced the libumem tool. See Find Leaks with the libumem Tool.

## Find Leaks with the dbx Debugger

The dbx debugger includes the Runtime Checking (RTC) functionality, which can find leaks. The dbx debugger is part of Oracle Solaris Studio and also available for Linux.

The following example shows a sample dbx session.

```
$ dbx ${java home}/bin/java
Reading java
Reading ld.so.1
Reading libthread.so.1
Reading libdl.so.1
Reading libc.so.1
(dbx) dbxenv rtc inherit on
(dbx) check -leaks
leaks checking - ON
(dbx) run HelloWorld
Running: java HelloWorld
(process id 15426)
Reading rtcapihook.so
Reading rtcaudit.so
Reading libmapmalloc.so.1
Reading libgen.so.1
Reading libm.so.2
Reading rtcboot.so
Reading librtc.so
RTC: Enabling Error Checking ...
RTC: Running program...
dbx: process 15426 about to exec("/net/bonsai.sfbay/export/home2/user/ws/
j2se/build/solaris-i586/bin/java")
dbx: program "/net/bonsai.sfbay/export/home2/user/ws/j2se/build/solaris-
i586/bin/java"
just exec'ed
dbx: to go back to the original program use "debug $oprog"
RTC: Enabling Error Checking ...
RTC: Running program...
t@1 (l@1) stopped in main at 0x0805136d
0x0805136d: main
                       :
                                pushl
                                          %ebp
```



```
(dbx) when dlopen libjvm { suppress all in libjvm.so; }
(2) when dlopen libjvm { suppress all in libjvm.so; }
(dbx) when dlopen libjava { suppress all in libjava.so; }
(3) when dlopen libjava { suppress all in libjava.so; }
(dbx) cont
Reading libjvm.so
Reading libsocket.so.1
Reading libsched.so.1
Reading libCrun.so.1
Reading libm.so.1
Reading libnsl.so.1
Reading libmd5.so.1
Reading libmp.so.2
Reading libhpi.so
Reading libverify.so
Reading libjava.so
Reading libzip.so
Reading en US.ISO8859-1.so.3
hello world
hello world
Checking for memory leaks...
Actual leaks report (actual leaks:
                                        27 total size:
46851 bytes)
          Num of Leaked
                           Allocation call stack
 Total
         Blocks Block
  Size
                  Address
_____
              4 –
    44376
                           calloc < zcalloc
     1072
              1 0x8151c70 nss XbyY buf alloc < get pwbuf <
getpwuid <
                             GetJavaProperties <
Java java lang System initProperties <
                             0xa740a89a< 0xa7402a14< 0xa74001fc
      814
               1 0x8072518 MemAlloc < CreateExecutionEnvironment <
main
      280
             10
                      _
                            operator new < Thread::Thread
               1 0x8072498 strdup < CreateExecutionEnvironment <
      102
main
               1 0x81697f0 calloc <
       56
Java java util zip Inflater init < 0xa740a89a<
                             0xa7402a6a< 0xa7402aeb< 0xa7402a14<
0xa7402a14< 0xa7402a14
       41
               1 0x8072bd8 main
                1 0x8072c58 SetJavaCommandLineProp < main
       30
       16
               1 0x806f180 setlocale < GetJavaProperties <
                             Java java lang System initProperties <
0xa740a89a< 0xa7402a14<
                             0xa74001fc< JavaCalls::call helper <</pre>
os::os exception wrapper
       12
               1 0x806f2e8 operator new <
instanceKlass::add dependent nmethod <</pre>
                             nmethod::new nmethod <</pre>
ciEnv::register method <
```

				Compile::Compile #Nvariant 1 <
C2Compi	ler::comp	bile	_method <	
				CompileBroker::invoke_compiler_on_method < CompileBroker::compiler_thread_loop
	12	1	0x806ee60	CheckJvmType < CreateExecutionEnvironment <
main				
	12	1	0x806ede8	<pre>MemAlloc &lt; CreateExecutionEnvironment &lt; main</pre>
	12	1	0x806edc0	main
	8	1	0x8071cb8	_strdup < ReadKnownVMs <
CreateE	xecutionE	Invi	ronment < m	ain
	8	1	0x8071cf8	_strdup < ReadKnownVMs <
CreateE	xecutionE	Invi	ronment < m	ain

The output shows that the dbx debugger reports memory leaks if memory is not freed at the time the process is about to exit. However, memory that is allocated at initialization time and needed for the life of the process is often never freed in native code. Therefore, in such cases, the dbx debugger can report memory leaks that are not really leaks.

#### Note:

The previous example used two suppress commands to suppress the leaks reported in the virtual machine: libjvm.so and the Java support library, libjava.so.

## Find Leaks with the libumem Tool

First introduced in the Oracle Solaris 9 operating system update 3, the libumem.so library, and the modular debugger mdb can be used to debug memory leaks.

Before using libumem, you must preload the libumem library and set an environment variable, as shown in the following example.

```
$ LD_PRELOAD=libumem.so
$ export LD_PRELOAD
$ UMEM_DEBUG=default
$ export UMEM_DEBUG
```

Now, run the Java application, but stop it before it exits. The following example uses truss to stop the process when it calls the exit system call.

```
$ truss -f -T exit java MainClass arguments
```

At this point you can attach the mdb debugger, as shown in the following example.

```
$ mdb -p pid
>::findleaks
```

The ::findleaks command is the mdb command to find memory leaks. If a leak is found, then this command prints the address of the allocation call, buffer address, and nearest symbol.



It is also possible to get the stack trace for the allocation that resulted in the memory leak by dumping the <code>bufctl</code> structure. The address of this structure can be obtained from the output of the ::findleaks command.

See analyzing memory leaks using libumem for troubleshooting the cause for a memory leak.

## 4

## Troubleshoot Performance Issues Using Flight Recorder

Identify performance issues with a Java application and debug these issues using recordings from Flight Recorder.

To learn more about creating a recording with Flight Recorder, see Produce a Flight Recording.

The data provided by Flight Recorder helps you investigate performance issues. No other tool gives as much profiling data without skewing the results with its own performance overhead. This chapter provides information about performance issues that you can identify and debug using data from Flight Recorder.

This chapter contains the following sections:

- Flight Recorder Overhead
- Use JDK Mission Control to Find Bottlenecks
- Use JDK Mission Control to Debug Garbage Collection Issues
- Use JDK Mission Control to Debug Synchronization Issues
- Use JDK Mission Control to Debug I/O Issues
- Use JDK Mission Control to Monitor Code Execution Performance

## Flight Recorder Overhead

When you measure performance, it is important to consider any performance overhead added by Flight Recorder. The overhead will differ depending on the application. If you have any performance tests set up, you can measure if there is any noticeable overhead on your application.

The overhead for recording a standard time fixed recording (profiling recording) using the default settings is less than two percent for most applications. Running with a standard continuous recording generally has no measurable performance effect.

Using Heap Statistics event, which is disabled by default, can cause significant performance overhead. This is because enabling Heap Statistics triggers an old garbage collection at the beginning and the at end of the test run. These old GCs give some extra pause times to the application, so if you are measuring latency or if your environment is sensitive to pause times, do not run with Heap Statistics enabled. Heap Statistics are useful when debugging memory leaks or when investigating the live set of the application. For more information, see Use JDK Mission Control to Debug Memory Leak.

#### Note:

For performance profiling use cases, heap statistics may not be necessary.



## Use JDK Mission Control to Find Bottlenecks

You can use JMC to find application bottlenecks.

One way to find out the application bottlenecks is to analyze the **Automated Analysis Results** page. This page provides comprehensive automatic analysis of flight recording data.

Open the **Threads** page in the **Java Application** page. The **Threads** page contains the following information:

- A graph that plots live thread usage by the application over time.
- A table with all live threads used by the application.
- Stack traces for selected threads.

Here is a sample figure of a recording, which shows a graph with thread details.



Figure 4-1 Bottlenecks - Threads - Graph

In the graph, each row is a thread, and each thread can have several lines. In the figure, each thread has a line, which represents the Java Application events that were enabled for this recording. The selected Java Application events all have the important property that they are all thread-stalling events. Thread stalling indicates that the thread was not running your application during the event, and they are all duration events. The duration event measures the duration the application was not running.

In the graph, each color represents a different type of event. For example:

- Yellow represents Java Monitor Wait events. The yellow part is when threads are waiting for an object. This often means that the thread is idle, perhaps waiting for a task.
- Salmon represents the Java Monitor Blocked events or synchronization events. If your Java application's important threads spend a lot of time being blocked, then that means that a critical section of the application is single threaded, which is a bottleneck.



- Red represents the Socket Reads and Socket Writes events. Again, if the Java
  application spends a lot of time waiting for sockets, then the main bottleneck may be in
  the network or with the other machines that the application communicates.
- *Green* represents parts that don't have any events. This part means that the thread is not sleeping, waiting, reading to or from a socket, or not being blocked. In general, this is where the application code is run. If your Java application's important threads are spending a lot of time without generating any application events, then the bottleneck in the application is the time spent executing code or the CPU itself.

#### Note:

For most Java Application event types, only events longer than 20 ms are recorded. (This threshold can be modified when starting the flight recording.) The areas may not have recorded events because the application is doing a lot of short tasks, such as writing to a file (a small part at a time) or spending time in synchronization for very short amounts of time.

The **Automated Analysis Results** page also shows information about garbage collections. To see if garbage collections may be a bottleneck, see the next topic about garbage collection performance.

## Use JDK Mission Control to Debug Garbage Collection Issues

You can use JMC to debug garbage collections (GC) issues.

Tuning the HotSpot Garbage Collector can have a big effect on performance. See Garbage Collection Tuning Guide for general information.

Take a profiling flight recording of your running application. Do not include the heap statistics, as that will trigger additional old garbage collections. To get a good sample, take a longer recording, for example one hour.

Open the recording in JMC. Look at the **Garbage Collections** section in the **Automated Analysis Results** page. Here is a sample figure of a recording, which provides a snapshot of garbage collection performance during runtime.

🚰 JDK Mission Control		- 0	×
<u>Eile Edit N</u> avigate <u>W</u> indow <u>H</u> elp			
🖬 JVM Browser 🔡 Outline 🛛 🚳 🕯			60
😸 Automated Analysis Results	△ w Automated Analysis Results		
<ul> <li>Java Application</li> </ul>		5 IS	< 1 mm
Threads	Java Application	& JVM Internals	1
<ul> <li>Memory</li> </ul>	- ouve / opplication		. 1
Live Objects	34 Threads Allocating	+ Stackdepth Setting +	
Lock Instances	42 Parallel Threads	+ Garbage Collections	
+≣ File I/O		100 G1/CMS Full Collection -	e
- Socket I/O	<ul> <li>Memory</li> </ul>		
Method Profiling	42 Allocated Classes	At least one Full, Stop-The-World Garbage Collection occurred during this recording. For the CMS and G1 collectors, Full GC events are a strong negative	
* Exceptions	88 Heap Content	performance indicator. Tunable GC parameters can be used to allow the collector	- 1
Thread Dumps	Most of the heap was used by only a few classes. If the heap usage needs	to operate in concurrent mode, avoiding Stop-The-World pauses and increasing GC and application performance.	
<ul> <li>W Internals</li> </ul>	reduced, then this would be a good place to start.	Co and approach performance.	61
Garbage Collections	Live Objects	71 GCs Caused by Heap Inspection - The JVM performed 8 heap inspection garbage collections. Performing heap	6
C Configuration	26 Heap Live Set Trend	inspection garbage collections may be a problem since they usually take a lot of	
> @ Compilations	The live set on the heap seems to increase with a speed of about 12.2 H	KiB per	
Class Loading WM Operations	second during the recording. An analysis of the reference tree found 1 leak candidates.	Environment	
TLAB Allocations	<ul> <li>java.util.Hashtable\$Entry[6534/98303]</li> <li>The candidate is referenced by this chain.</li> </ul>	© Processes	.
✓ <sup>I</sup> Environment	<ul> <li>java.util.Hashtable.table (Stack Variable : Threads (Threads)</li> </ul>	ead 81 Competing CPU Ratio Usage -	
G Processes	V Name: main))	An average CPU load of 45 % was caused by other processes during 32.415 s	1
Properties 💷 🔤 Results		starting at 7/7/20, 5:05:24 PM.	
Field Value 0 events		The application performance can be affected when the machine is under heavy load and there are other processes that use CPU or other resources on the same committer. To profile accounting the resource that there its an other the same the same set of the same set of t	ŀ

#### Figure 4-2 Automated Analysis Results - Garbage Collections



You can observe from the figure that there is a Full GC event. This is indicative of the fact that application needs more memory than what you have allocated.

For further analysis, open the **Garbage Collections** page under the **JVM Internals** page to investigate the overall performance impact of the GC. Here is a sample figure of a recording, which shows a graph with GC pauses.

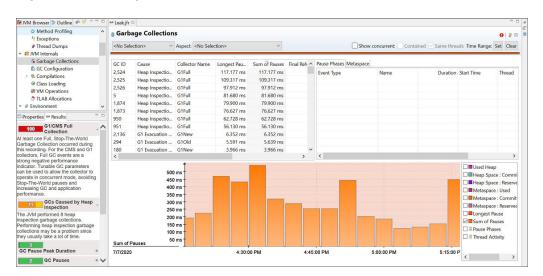


Figure 4-3 Garbage Collection Performance - GC Pauses

From the graph look at the **Sum of Pauses** from the recording. The **Sum of Pauses** is the total amount of time that the application was paused during a GC. Many GCs do most of their work in the background. In those cases, the length of the GC does not matter and what matters is how long the application actually had to stop. Therefore, the **Sum of Pauses** is a good measure for the GC effect.

The main performance problems with garbage collections are usually either that individual GCs take too long, or that too much time is spent in paused GCs (total GC pauses).

When an individual GC takes too long, you may need to change the GC strategy. Different GCs have different trade-offs when it comes to pause times verses throughput performance. See Behavior-Based Tuning.

In addition, you may also need to fix your application so that it makes less use of finalizers or semireferences. See Monitor the Objects Pending Finalization and Finalization and Weak, Soft, and Phantom References in *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide* for information about detecting and migrating from finalization.

If the application spends too much time paused, you can look into different ways to overcome this. One way is to increase the Java heap size. Look at the **GC Configuration** page to estimate the heap size used by the application, and change the initial heap size and maximum heap size to a higher value. The bigger the heap, the longer time it is between GCs. Watch out for any memory leaks in the Java application, because that may cause more frequent GCs until an OutOfMemoryError is thrown. For more information, see Use JDK Mission Control to Debug Memory Leak. Another way to reduce the GC cycles is to allocate fewer temporary objects. In the **TLAB Allocations** page, look at how much memory is allocated over the course of the recording. Small objects are allocated inside TLABs, and large objects are allocated outside TLABs. Often, the majority of allocations happen inside TLABs. Lastly, to reduce the need of GCs, decrease the allocation rate. Select the **TLAB Allocations** page and then look at the allocation sites that have the most memory pressure. You can either view it per class or thread to see which one consumes the most allocation.

Some other settings may also increase GC performance of the Java application. See Garbage Collection Tuning Guide in the Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide to discuss GC performance.

## Use JDK Mission Control to Debug Synchronization Issues

You can use JMC to debug Java Application synchronization issues.

Open the flight recording in JMC and look at the **Automated Analysis Results** page. Here is a sample figure of a recording, which shows threads that are blocked on locks.

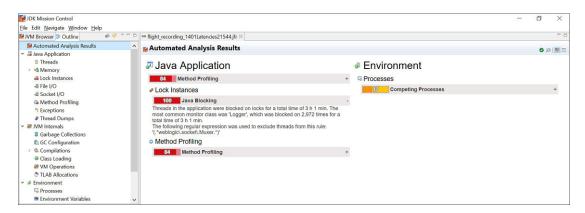


Figure 4-4 Synchronization Issue - Automated Analysis Results Page

Focus on the **Lock Instances** section of the page, which is highlighted in red. This is indicative of a potential problem. You can observe that there are threads that are blocked on locks.

For further analysis, open the **Lock Instances** page. Here is a sample figure of a recording, which shows the thread that is blocked on locks the most and the *stack trace* of the thread waiting to acquire the lock.



🛃 JVM Browser 🐉 Out	ine o	1 🧳 🗸 🖛 🗖	← flight_recording_1401Latencies21544.jfr 🕸					
Mutomated Analy		^	Lock Instances					0
<ul> <li>Java Application</li> <li>Threads</li> </ul>			<no selection=""></no>	~ ~ ~			Chow concurrent	Contained Same threads
> S Memory			<ul> <li>Aspect. <no li="" selection<=""> </no></li></ul>				Show concurrent.	Contained 💌 Same triteads
lock Instance	s		Monitor Class	at i sai	cked Time	Distinct Threads	Count	
- File I/O								
-8 Socket I/O			<sup>O</sup> Logger		3 h 1 min	20	2,972	
G Method Profil	ing							
<sup>1</sup> Exceptions								
Thread Dump	s							
<ul> <li>WM Internals</li> </ul>								
Garbage Colle	ections		Monitor Address	Total Blo	cked Time	Distinct Threads	Count	
C Configurat			⊠0x1EC73552088		3 h 1 min	20	2,972	
> 9. Compilations								
Generation Class Loading								
WM Operation								
TLAB Allocation	ons							
✓			Thread	Total Blocked Time	2	Count		^
Processes			Worker Thread 1	9 min 36 s		82		
Environment \	Variables	~	PWorker Thread 8	9 min 36 s		82		
🗆 Properties 💠 🚥 Resu	ults	<b>1</b> × • •	Morker Thread 18	9 min 36 s		82		
Field	Value	^	Worker Thread 2	9 min 35 s		82		
Event Type	Java Monit		▶ Worker Thread 7	9 min 165 ms		166		~
Start Time	7/17/2020,							k X K = 0 + 7 *
Duration	71.250 ms		Stack Trace					K N K   E   🔿 🍖 🗸
Send Time	7/17/2020,		Stack Trace			ount		
P Event Thread	Too many v		toid Logger.log(String)		2	972		
Monitor Class	Logger		† void Worker.run()			972		
Previous Monitor	Too many v		1 void java.lang.Thread.run()		2	972		
Monitor Address	0x1EC7355							
	2.972 events	~						

Figure 4-5 Synchronization Issue - Lock Instance

You can notice that threads in the application were blocked on locks for a total time of 3 hours. The most common monitor class in contention was Logger, which was blocked 2972 times.

Typically, logging is an area that can be a bottleneck in applications. In this scenario, the blocking events all seem to be due to calls to the log method. You can review and make required code changes to fix this issue.

## Use JDK Mission Control to Debug I/O Issues

You can diagnose I/O issues in an application by monitoring the **Socket I/O** or the **File I/O** pages in JMC.

When a Java application spends a lot of time either in **Socket Read**, **Socket Write**, **File Read**, or **File Write**, then I/O or networking can cause bottleneck. To diagnose I/O issues in applications, open the **Socket I/O** page under the **Java Application** page in the **Automated Analysis Results** page. Here is a sample figure of a recording, which shows Socket I/O details.



4 # Tile VO       0ededede:1       195:/47 ms       2       2         • Socket VO       0 Method Profiling       115:/47 ms       2       2         • Method Profiling       127.0.0.1       6.084 s       269       245       24       46.11 k         • Method Profiling       127.0.0.1       6.084 s       269       245       24       46.11 k         • Method Profiling       • Count       Remote Port       Total I/O Time       Count       Read Count       Write Count         • Goodspecifications       6 Grabage Collections       6 Grabage Collections       Start Time       Remote Port       Duration Bytes Read       Bytes Writen Thread       End of y24/15.339:12 PM       10.161:190:213       58.454       20.037 ms       92.98       RMTCP Co         • Grasse       • Environment       • Start Time       Remote Address       Remote Port       Duration Bytes Read       Bytes Writen Thread       End of y24/15.339:12 PM       10.161:190:213       58.6493       349.745 ms       81.8       ACTIVE Ex         • Proventies       • Environment       • Total I/O Start Time       Start Time       Start Time       Start Time       Start Time       Start Time       Total Time Start Time       Total Time Start Time       Start Time       Start Time       Total Time Star	JVM Browser 🗄 Outline	ali 💛 🗢 🗆 🗆	∞ wldf.jfr ≋								- C
In took instances       Interview       Intervi	Java Application	^		V Aspect: <no selection=""></no>	~		Show	v concurrent: Contained 🗹	Same threads Tim		R 🗆
• Method Profiling       • Method Profiling       2 min 20 s       273       265       8       153.00         • Method Profiling       • Exceptions       6.084 s       269       245       24       46.11 kD         • Method Profiling       • Exceptions       • Configuration       • Sect Time Remote Address       Remote Port       Duration Bytes Read       Bytes Written Thread       End of 9.2415, 33.93.11 PM       10.161.190.213       58.458       20.037 ms       92.86       RMI CP Co         • Of List Ad Allocations       • Environment       • Environment       • Seck Tarce       • Environment       • • • • • • • • • • • • • • • • • • •	Lock Instances     Elic I/O     Socket I/O     Method Profiling     Exceptions     # Thread Dumps					ad Count	^ Write Count	Bytes Read	Bytes Written 3.47 KiB		
* 1 Tread Dumps       Remote Port       Total /O Time       Count       Read Count       Write Count         * & 3 WM Internals       Compliations       Compliations       Compliations       End count       Write Count       End count       Write Count         * @ Compliations       Compliations       Sock Trans       929 8       RMI TCP Co       End count       End coun									15.3 KiB	7.72 KiB 21.4 KiB	
A Minima Durations     Compliations     Compliation     Compliatin     Compliation     Compliation     Compliation     Compliatio						Read Count	245		46.1 KiB Bytes R		<ul> <li>•</li> </ul>
6 Garbage Collections       Immeline Durations Event Log         0 G. Compliantions       9 Carbinations         9 G. Compliantions       9 Start Time Remote Address         9 G. Compliantions       9 Start Time Remote Address         9 Comparisons       9 Start Time Remote Address         9 Comparisons       9 Start Time Remote Address         9 Comparisons       9 Start Time Remote Address         9 Properties       9 Start Time Remote Address         9 Properties       8 Boults         9 Properties       Results         9 Conscisses       Image: Start Time Remote Address         9 Properties       Results       0 Total Start Time Remote Address         9 Address tecorde socket Write Peak Duration       1         10 Total Address tecorde socket Write Peak Duration       1         10 Total Address tecorde socket Write Poak Duration       1         10 Total Address tecorde socket Write Poak Duration       1         10 Total Address tecorde socket Write Poak Duration       1         10 Total Address that we compare the total address tecorde socket Muscoket/Muscof NIOCoutputStream-writeInd(OutputStream-writeInd(Outp			A CONTRACTOR OF CONTRACTOR	lotal I/O Time	Count	Read Count		Write Count	Bytes R	2775).	> ~
Start Time Remote Address Remote Port Duration Bytes Read Bytes Written Thread End of 9 G. Compliations G. Cass Loading 9/24/15, 33:83 TM 10.161.190.213 58.698 349.745 ms 929 8 RM TCP Co 9/24/15, 33:83 TM 10.161.190.213 58.699 349.745 ms 818 [ACTIVE] Ex 9/24/15, 33:93.14 PM 10.161.190.213 58.679 349.745 ms 818 [ACTIVE] Ex 9/24/15, 33:93.14 PM 10.161.190.213 58.679 349.745 ms 818 [ACTIVE] Ex 9/24/15, 33:93.14 PM 10.161.190.213 58.679 349.745 ms 818 [ACTIVE] Ex 9/24/15, 33:93.14 PM 10.161.190.213 58.716 21.251 ms 818 [ACTIVE] Ex 9/24/15, 33:93.14 PM 10.161.190.213 58.716 21.251 ms 818 [ACTIVE] Ex 9/24/15, 33:93.14 PM 10.161.190.213 58.716 21.251 ms 818 [ACTIVE] Ex 9/24/15, 33:93.14 PM 10.161.190.213 58.716 21.251 ms 818 [ACTIVE] Ex 9/24/15, 33:93.14 PM 10.161.190.213 58.716 21.251 ms 818 [ACTIVE] Ex 9/24/15, 33:93.14 PM 10.161.190.213 58.716 21.251 ms 818 [ACTIVE] Ex 9/24/15, 33:93.14 PM 10.161.190.213 58.716 21.251 ms 818 [ACTIVE] Ex 9/24/15, 33:93.14 PM 10.161.190.213 58.716 21.251 ms 818 [ACTIVE] Ex 9/24/15, 33:93.14 PM 10.161.190.213 58.716 21.251 ms 818 [ACTIVE] Ex 9/24/15, 33:93.14 PM 10.161.190.213 58.716 21.251 ms 818 [ACTIVE] Ex 9/24/15, 33:93.14 PM 10.161.190.213 58.716 21.251 ms 818 [ACTIVE] Ex 9/24/15, 33:93.14 PM 10.161.190.213 58.716 21.251 ms 818 [ACTIVE] Ex 9/24/15, 33:93.14 PM 10.161.190.213 58.716 21.251 ms 818 [ACTIVE] Ex 9/24/15, 33:93.14 PM 10.161.190.213 58.716 21.251 ms 818 [ACTIVE] Ex 9/24/15, 33:93.14 PM 10.161.190.213 59.716 21.251 ms 818 [ACTIVE] Ex 9/24/15, 33:93.14 PM 10.161.190.213 59.716 21.251 ms 818 [ACTIVE] Ex 9/24/15, 33:93.14 PM 10.161.190.213 [ACTIVE] Ex 9/24/15, 33:93.14 PM 10.161.190.213 [ACTIVE] Ex 9/24/15, 33:93.14 PM 10.161.190.213 [ACTIVE] Ex 9/24/15, 32:93.14 PM 10.161.190.213 [ACTIVE] Ex 9/24/15, 32:93.14 PM 10.161.190.213 [ACTIVE] Ex 9/24/15, 32:93.14 PM 10.161.190.214 [ACTIVE] Ex 9/24/15, 32:93.14 PM 10.161.190.214 [ACTIVE] Ex 9/24/15, 32:93.14 PM 10.161.190.214 [ACTIVE] Ex				ent log							,
#/W Operations       9/24/15,3:39:14 PM 10.161.190.213       58,716       21.251 ms       81.8 [ACTIVE] Ex         # Drocesses       Environment            81.8 [ACTIVE] Ex          # Processes       Environment Variables             81.8 [ACTIVE] Ex         * 0 Properties ** Results                   81.8 [ACTIVE] Ex   <	> @ Compilations		Start Time 9/24/15, 3:38:31 PM	Remote Address 10.161.190.213	58,548	29.037 ms	Bytes Read	929 B RMI TCP		n Ti	im ^
Processes     Environment     Processes     Environment Variables     Count     Stack Trace     Sack Trace     Sack Trace     Sack Trace     Sack Trace     Sack Trace     Count     Int unio ch.Socket(hannellmpLwrite(8)rt8uffer)     void weblogic.sorket.NIOSocket(huser\$NIOOutputStream.writeInternal(8)rt9uf8uffer)     void weblogic.sorket.NIOSocket(huser\$NIOOutputStream.writeInternal(8)rt9uf8uffer)     void weblogic.sorket.INIOSocket(huser\$NIOOutputStream.writeInternal(8)rt9uf8uffer)     void weblogic.sorket.INIOSocket(huser\$NIOOutputStream.writeInternal(8)rt9uf8uf9uf8uffer)     void weblogic.sorket.INIOSocket(huser\$NIOOutputStream.writeInternal(8)rt9uf8uf9uf8uf4eder(funkedData     void weblogic.sorket.Internal.Servlet(9)rt9uf8uf8uf1eder(5) 1	VM Operations     TLAB Allocations     Allocations     A Environment     Processes		The Court Strength Court of the								~
Environment Variables     Stack Trace			<								>
Properties       Int sun nio.ch. Socket(DannelimpLwrite)8yteBuffer)       Int sun nio.ch. Socket(DannelimpLwrite)8yteBuffer)         20       Socket Write Peak Duration       Int sun nio.ch. Socket(DannelimpLwrite)8yteBuffer)         10 int sun nio.ch. Socket(DannelimpLwrite)8yteBuffer)       Void weblogic.socket.NIOSocketMuxerSNIOOutputStream.writeInternal[8yteBuffer)         10 int sun nio.ch. Socket(DannelimpLwrite)8yteBuffer)       Void weblogic.socket.NIOSocketMuxerSNIOOutputStream.writeInternal[8yteBuffer)         10 int sun store internet servere       Void weblogic.socket.NIOSocketMuxerSNIOOutputStream.writeInternal[8yteBuffer)         10 int sun store internet servere       Void weblogic.sorket.Internal.ServereUtstream.intelinternal[8yteBuffer)         10 int sun store       Int weblogic.servet.Internal.ServereUtstream.intelintereleaders(ServeteOutputStream.writeIntereleaders)         10 int sun store       Void weblogic.servet.Internal.ServereUtstream.inteleaders(ServeteInterenal]         10 void weblogic.servet.Internal.ServereUtstream.impl.writeIedeaders(ServeteInterenal]       Void weblogic.servet.Internal.ServereUtstream.impl.writeIedeaders(ServeteInterenal]			Stack Trace						кя	K 🔢 🛛 🖻 💈	~ ~ (
20       Socket Write Peak Duration <ul> <li>void weblogic.socket.NIOSocketMuxer\$NIOOutputStream.writeInternal@yteBuffer)</li> <li>to objekt in the socket</li> <li>void weblogic.socket.NIOSocketMuxer\$NIOOutputStream.writeInternal@yteBuffer)</li> <li>void weblogic.socket.NIOSocketMuxer\$NIOOutputStream.mriteInternal@yteBuffer)</li> <li>void weblo</li></ul>	Properties as Results	° 0		deal Channel and a state (Deale Dealface)			Count				1
48,746 ms to write 81 B to the host at 0,011 ms to write 81 B to the host at 0,011 ms to write 30 ms to write	28 Socket Write Peak D	Duration -			ream.writeInternal(Byte	Buffer)	1				
Junid weblasic could internal Could Output StreamIngl condeloadore()	The longest recorded socket write took 349,745 ms to write 81 B to the host at 10,161,190,213. Note that there are some socket write patterns with high duration writes that we consider to be normal and are therefore excluded. Such patterns include JMX FMI communication.		void weblogicsutisi.o.ChunkedOutputStream.writeTo(OutputStream)       1         void weblogic.servlet.internal.ServletOutputStream.writeTeader(ChunkedDatat       1         intt weblogic.servlet.internal.ServletBeaders(ServletOutputStream)       1         void weblogic.servlet.internal.ServletBeaders(ServletOutputStream)       1         void weblogic.servlet.internal.ServletBeaders(ServletOutputStream)       1								
Void web/gc.servet.internal.ServetCouputStreamImp.Enduateservet     void web/gc.servet.internal.ServetCouputStreamImp.Enduate	10 Socket Read Peak Duration						1				
void weblogic.servlet.internal.ServletResponseImpl.flushBuffer()     void weblogic.wsee.connection.transport.servlet.HttpServerTransport.confirmOnew							1				

#### Figure 4-6 Socket I/O - Java Application

The figure shows that for the application the longest recorded socket write took 349.745 ms to write 81 B to the host.

File or networking I/O issues are diagnosed in a similar fashion. Look at the files read from or written to the most, then see each file read/write and the time spent on I/O.

By default, the **Socket I/O** page lists events with a duration longer than 10 ms. When starting a flight recording, you can lower the file I/O Threshold or socket I/O Threshold to gather more data, but this could potentially have a higher performance overhead.

# Use JDK Mission Control to Monitor Code Execution Performance

You can use JMC to monitor the code execution performance.

When there are not a lot of Java Application events, it could be that the main bottleneck of your application is the running code. In such scenarios, look at the **Method Profiling** section of the **Automated Analysis Results** page. Here is a sample figure of a recording, which indicates that there is value in optimizing certain methods.

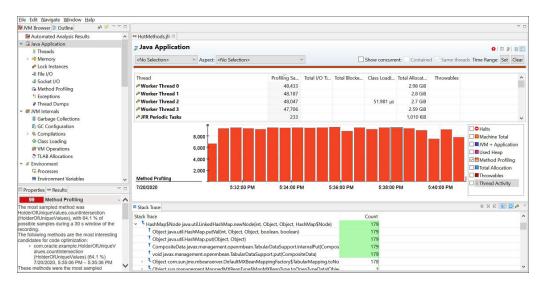


JDK Mission Control				 ٥	×
Eile Edit Navigate Window Help	an HotMethods.ifr ≅				-
Automated Analysis Results     Automated Analysis Results     Automated Analysis Results	Mutomated Analysis Results			0	10   📰
Threads     Memory	Java Application		Environment		1
Lock Instances	35 Threads Allocating	+	S Processes		
-8 File I/O	23 High JVM CPU Load	+	37 Competing Processes		+
+8 Socket I/O G Method Profiling	99 Method Profiling	+	, ,		
1 Exceptions	⊲ Memory				
# Thread Dumps	53 Allocated Classes	+			
<ul> <li>WM Internals</li> <li>Garbage Collections</li> </ul>	55 Primitive To Object Conversion	+			
C Configuration	Method Profiling				
> 9. Compilations	Method Profiling				
Class Loading     W Operations	The most sampled method was HolderOfUniqueValues.countIntersection				
TLAB Allocations	(HolderOfUniqueValues), with 64.1 % of the maximum possible samples during a 30 s window of the recording, and 99.5 % of the actual samples.				
Y @ Environment	The following methods are the most interesting candidates for code optimization: • com.oracle.example.HolderOfUniqueValues.countIntersection				
Processes Environment Variables	(HolderOfUniqueValues) (99.5 % of samples) 7/20/2020, 5:35:06 PM – 5:35:36 PM				- 1
Properties 😁 Results 👘 🗖	the recording. The percentage shown for each method tens now many execution	ſ			
There are no results associated with this page.	samples it was seen in compared to the maximum possible number of samples during that window.				~

Figure 4-7 Code Execution Performance - Automated Analysis Results Page

Now, open the **Java Application** page. Here is a sample figure of a recording, which shows the **Method Profiling** graph and the stack traces.





You can observe that the stack trace view shows the aggregated stack traces of any selection in the editor and also the stack traces for the profiling samples. In the figure, you can notice that one of these methods has a lot more samples than the others. This means that the JVM has spent more time executing that method relative to the other methods.

To identify which method would be the one to optimize to improve the performance of the application, open the **Method Profiling** page. Here is a sample figure of a recording, which shows the method that needs to be optimized.



👹 JVM Browser 🔠 Outline 🛛 🖉 🖤 🖤					• c		
Automated Analysis Results	^	∞ HotMethods.ifr ≅					
✓ Java Application ■ Threads		o Method Profiling		•   •			
> 📸 Memory		<no selection=""></no>	~		Show concurrent: Contained Same threads		
Lock Instances							
-B File I/O		Top Package	Count				
+≣ Socket I/O		java.util	191,444				
G Method Profiling		java.lang	728				
* Exceptions		jdk.jfr.internal	232				
Thread Dumps		com.oracle.example	199				
<ul> <li>IVM Internals</li> </ul>		idk.ifr.internal.handlers	2				
Garbage Collections		jdk.jfr.events	1				
C Configuration			~				
> @ Compilations		Top Class	Count				
Class Loading		© com.oracle.example.HolderOfUniqueValues	186				
WM Operations		<sup>©</sup> com.oracle.example.Worker	10				
TLAB Allocations Allocations		<sup>©</sup> com.oracle.example.WorkEvent	3				
Processes							
Processes Environment Variables							
	~						
Properties and Results	• •	Stack Trace			K 7 K = 0 P -		
	~			-	n n n ( <u>m</u> ) v		
The most sampled method was		Stack Trace		Count			
HolderOfUniqueValues, countintersection (HolderOfUniqueValues), with 64.1 % of possible samples during a 30 s window of the recording. The following methods are the most interesting candidates for code optimization: • com.oracle.example.HolderOfUniqueV alues.countintersection		<ul> <li>t int com.oracle.example.HolderOfUniqueValues.countIn</li> </ul>	tersection(HolderOfUniqueValues)	178			
		void com.oracle.example.Worker.run()		178			
		<sup>1</sup> void java.lang.Thread.run()		178			
		void com.oracle.example.Worker.run()	10				
		void com.oracle.example.HolderOfUniqueValues.initiali	\$				
		void com.oracle.example.HolderOfUniqueValues. <init></init>	3				
(HolderOfUniqueValues) (64.1 %) 7/20/2020, 5:35:06 PM - 5:35:36 PM		void com.orade.example.WorkEvent.begin()		2			
These methods were the most sampled	V	void com.oracle.example.WorkEvent. <init>()</init>		1			

Figure 4-9 Code Execution Performance - Method Profiling

As you can observe, in the stack trace view, the most sampled method was

HolderOfUniqueValues.countIntersection(). You can review and make required code changes to optimize this method to effectively improve the performance of the application.



# Part II Debug JVM Issues

This part describes causes and various debugging techniques for the following topics.

Troubleshoot System Crashes

Provides guidance about specific procedures for troubleshooting system crashes.

- Troubleshoot Process Hangs and Loops
  - Provides guidance about specific procedures for troubleshooting hanging or looping processes.
- Handle Signals and Exceptions

Provides guidance about signal and exception handling by Java HotSpot Server VM.



# 5 Troubleshoot System Crashes

This chapter presents information and guidance about some specific procedures for troubleshooting system crashes.

A crash, or fatal error, causes a process to terminate abnormally. There are various possible reasons for a crash. For example, a crash can occur due to a bug in the Java HotSpot VM, in a system library, in a Java SE library or an API, in application native code, or even in the operating system (OS). External factors, such as resource exhaustion in the OS can also cause a crash.

Crashes caused by bugs in the Java HotSpot VM or in the Java SE library code are rare. This chapter provides suggestions about how to examine a crash and work around some of the issues (if possible) until the cause of the bug is diagnosed and fixed.

In general, the first step with any crash is to locate the fatal error log. This is a text file that the Java HotSpot VM generates in the event of a crash. See Fatal Error Log for an explanation of how to locate this file, as well as a detailed description of the file.

This chapter contains the following sections:

- Determine Where the Crash Occurred
- Find a Workaround
- Microsoft Visual C++ Version Considerations

## Determine Where the Crash Occurred

Examples that demonstrate how the error log can be used to find the cause of the crash, and suggests some tips for troubleshooting the problem depending on the cause.

The error log header indicates the type of error and the problematic frame, while the thread stack indicates the current thread and stack trace. See Header Format.

The following are possible causes for the crash.

- Crash the Native Code
- Crash in the Compiled Code
- Crash in the HotSpot Compiler Thread
- Crash in the VM Thread
- Crash Due to Stack Overflow

## Crash the Native Code

Analyze the crash dump file or core file to identify if the crash occurred in the native code or the Java Native Interface (JNI) library code.

If the fatal error log indicates the problematic frame to be a native library, then there might be a bug in the native code or the Java Native Interface (JNI) library code. The crash could be



caused by something else, but analysis of the library and any core file or crash dump is a good starting place. Consider the extract in the following example from the header of a fatal error log.

```
# An unexpected error has been detected by HotSpot Virtual Machine:
#
# SIGSEGV (0xb) at pc=0x417789d7, pid=21139, tid=1024
#
# Java VM: Java HotSpot(TM) Server VM (6-beta2-b63 mixed mode)
# Problematic frame:
# C [libApplication.so+0x9d7]
```

In this case a SIGSEGV occurred with a thread executing in the library libApplication.so.

In some cases a bug in a native library manifests itself as a crash in Java VM code. Consider the crash in the following example where a JavaThread fails while in the thread in vm state (meaning that it is executing in Java VM code).

```
# An unexpected error has been detected by HotSpot Virtual Machine:
#
# EXCEPTION ACCESS VIOLATION (0xc0000005) at pc=0x08083d77, pid=3700,
tid=2896
# Java VM: Java HotSpot(TM) Client VM (1.5-internal mixed mode)
# Problematic frame:
# V [jvm.dll+0x83d77]
----- T H R E A D -----
Current thread (0x00036960): JavaThread "main" [ thread in vm,
id=2896]
:
Stack: [0x00040000,0x00080000), sp=0x0007f9f8, free space=254k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code,
C=native code)
V [jvm.dll+0x83d77]
C [App.dll+0x1047]
                          <======= C/native frame
j Test.foo()V+0
j Test.main([Ljava/lang/String;)V+0
v ~StubRoutines::call stub
V [jvm.dll+0x80f13]
V [jvm.dll+0xd3842]
V
  [jvm.dll+0x80de4]
V [jvm.dll+0x87cd2]
C [java.exe+0x14c0]
C [java.exe+0x64cd]
C [kernel32.dll+0x214c7]
 :
```

In this case, although the problematic frame is a VM frame, the thread stack shows that a native routine in App.dll has called into the VM (probably with JNI).



The first step to solving a crash in a native library is to investigate the source of the native library where the crash occurred.

- If the native library is provided by your application, then investigate the source code of your native library. A significant number of issues with JNI code can be identified by running the application with the -Xcheck:jni option added to the command line. See The -Xcheck:jni Option.
- If the native library has been provided by another vendor and is used by your application, then file a bug report against this third-party application and provide the fatal error log information.
- If the native library where the crash occurred is part of the Java Runtime Environment (JRE) (for example awt.dll, net.dll, and so forth), then it is possible that you encountered a library or API bug. If so, gather as much data as possible, and submit a bug or report, indicating the library name. You can find JRE libraries in the jre/lib or jre/bin directories of the JRE distribution. See Submit a Bug Report.

You can troubleshoot a crash in a native application library by attaching the native debugger to the core file or crash dump, if it is available. Depending on the OS, the native debugger is dbx, gdb, or windbg. See Native Operating System Tools.

## Crash in the Compiled Code

Analyze the fatal error log to identify if the crash occurred in the compiled code.

If the fatal error log indicates that the crash occurred in compiled code, then it is possible that you encountered a compiler bug that resulted in incorrect code generation. You can recognize a crash in compiled code if the type of the problematic frame is J (meaning a compiled Java frame). The following example shows such a crash.

```
# An unexpected error has been detected by HotSpot Virtual Machine:
#
#
  SIGSEGV (0xb) at pc=0x0000002a99eb0c10, pid=6106, tid=278546
#
#
 Java VM: Java HotSpot(TM) 64-Bit Server VM (1.6.0-beta-b51 mixed mode)
# Problematic frame:
#
 J org.foobar.Scanner.body()V
#
:
Stack: [0x0000002aea560000,0x0000002aea660000), sp=0x0000002aea65ddf0,
  free space=1015k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native
code)
J org.foobar.Scanner.body()V
```

[error occurred during error reporting, step 120, id 0xb]

#### Note:

A complete thread stack is not available. The output line "error occurred during error reporting" means that a problem arose trying to get the stack trace (this might indicate stack corruption).



It might be possible to temporarily work around the issue by switching the compiler or by excluding from compilation the method that provoked the crash.

See Working Around Crashes in the HotSpot Compiler Thread or Compiled Code.

## Crash in the HotSpot Compiler Thread

Analyze the fatal error log to identify if the crash occurred in the HotSpot compiler thread.

If the fatal error log output shows that the current thread is a JavaThread named CompilerThread0, CompilerThread1, or AdapterCompiler, then it is possible that you encountered a compiler bug. In this case, it might be necessary to temporarily work around the issue by switching the compiler (for example, by using the HotSpot Client VM instead of the HotSpot Server VM, or vice versa), or by excluding from compilation the method that provoked the crash.

See Working Around Crashes in the HotSpot Compiler Thread or Compiled Code.

## Crash in the VM Thread

Analyze the fatal error log to identify if the crash occurred in the VMThread.

If the fatal error log output shows that the current thread is a VMThread, then look for the line containing VM\_Operation in the THREAD section. A VMThread is a special thread in the HotSpot VM. It performs special tasks in the VM such as garbage collection (GC). If the VM\_Operation suggests that the operation is a GC, then it is possible that you encountered an issue such as heap corruption.

Beside a GC issue, it could be something else (such as a compiler or runtime bug) that leaves object references in the heap in an inconsistent or incorrect state. In this case, collect as much information as possible about the environment and try possible workarounds. If the issue is related to GC, then you might be able to temporarily work around the issue by changing the GC configuration.

See Working Around Crashes During Garbage Collection.

## Crash Due to Stack Overflow

A stack overflow in the Java language code will normally result in the offending thread throwing the <code>java.lang.StackOverflowError</code> exception.

On the other hand, C and C++ write beyond the end of the stack and cause a stack overflow. This is a fatal error that causes the process to terminate.

In the HotSpot implementation, Java methods share stack frames with C/C++ native code, namely user native code and the virtual machine itself. Java methods generate code that checks whether the stack space is available at a fixed distance towards the end of the stack so that the native code can be called without exceeding the stack space. The distance toward the end of the stack is called shadow pages. The size of the shadow pages is between 3 and 20 pages, depending on the platform. This distance is tunable, so that applications with native code needing more than the default distance can increase the shadow page size. The option to increase shadow pages is -XX:StackShadowPages=n, where *n* is greater than the default stack shadow pages for the platform.



If your application gets a segmentation fault without a core file or fatal error log file, see Fatal Error Log. Or if you application gets a STACK\_OVERFLOW\_ERROR on Windows or the message "An irrecoverable stack overflow has occurred," then this indicates that the value of StackShadowPages was exceeded, and more space is needed.

If you increase the value of StackShadowPages, you might also need to increase the default thread stack size using the -Xss parameter. Increasing the default thread stack size might decrease the number of threads that can be created, so be careful in choosing a value for the thread stack size. The thread stack size varies by platform from 256 KB to 1024 KB.

```
# An unexpected error has been detected by HotSpot Virtual Machine:
#
  EXCEPTION STACK OVERFLOW (0xc00000fd) at pc=0x10001011, pid=296, tid=2940
#
#
# Java VM: Java HotSpot(TM) Client VM (1.6-internal mixed mode, sharing)
# Problematic frame:
# C [App.dll+0x1011]
#
----- THREAD ------
Current thread (0x000367c0): JavaThread "main" [ thread in native, id=2940]
Stack: [0x00040000,0x00080000), sp=0x00041000, free space=4k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native
code)
C [App.dll+0x1011]
C [App.dll+0x1020]
C [App.dll+0x1020]
:
C [App.dll+0x1020]
C [App.dll+0x1020]
...<more frames>...
Java frames: (J=compiled Java code, j=interpreted, Vv=VM code)
j Test.foo()V+0
j Test.main([Ljava/lang/String;)V+0
v ~StubRoutines::call stub
```

You can interpret the following information from the above example.

- The exception is **EXCEPTION STACK OVERFLOW**.
- The thread state is \_thread\_in\_native, which means that the thread is executing native or JNI code.
- In the stack information, the free space is only 4 KB (a single page on a Windows system). In addition, the stack pointer (sp) is at 0x00041000, which is close to the end of the stack at 0x00040000.
- The printout of the native frames shows that a recursive native function is the issue in this case. The output notation ... <more frames>... indicates that additional frames exist but were not printed. The output is limited to 100 frames.



# Find a Workaround

Possible workarounds if a crash occurs with a critical application.

If a crash occurs with a critical application, and the crash appears to be caused by a bug in the HotSpot VM, then it might be desirable to quickly find a temporary workaround. If the crash occurs with an application that is deployed with the most recent release of the JDK, then the crash should be reported to Oracle.

#### Important:

Even if a workaround in this section successfully eliminates a crash, the workaround is **not** a fix for the problem, but merely a temporary solution. Place a support call or file a bug report with the original configuration that demonstrated the issue.

The following are three scenarios to find workarounds for system crashes.

- · Working Around Crashes in the HotSpot Compiler Thread or Compiled Code
- Working Around Crashes During Garbage Collection
- Working Around Crashes Caused by Class Data Sharing

# Working Around Crashes in the HotSpot Compiler Thread or Compiled Code

Possible workarounds if the crash occurred in the hotspot compiler thread.

If the fatal error log indicates that the crash occurred in a compiler thread, then it is possible (but not always the case) that you encountered a compiler bug. Similarly, if the crash is in compiled code, then it is possible that the compiler generated incorrect code.

In the case of the HotSpot Client VM (-client option), the compiler thread appears in the error log as CompilerThread0. With the HotSpot Server VM, there are multiple compiler threads, and these appear in the error log file as CompilerThread0, CompilerThread1, and AdapterThread.

Since the JDK 7u5 release, the HotSpot compiler is ignored by default. A commandline option is available to simulate the old behavior, which is useful when multiple methods were excluded. See notable bug fixes in JDK 7u5.

To exclude methods from being compiled by using a JVM flag instead of the .hotspot\_compile file, see -XX:CompileCommand in Advanced JIT Compiler Options in the Java Platform, Standard Edition Tools Reference.

The following example shows a fragment of an error log for a compiler bug that was encountered and fixed during development. The log file shows that the HotSpot Server VM is used, and the crash occurred in CompilerThread1. In addition, the log file shows



that the current CompileTask was the compilation of the java.lang.Thread.setPriority method.

```
# An unexpected error has been detected by HotSpot Virtual Machine:
#
:
# Java VM: Java HotSpot(TM) Server VM (1.5-internal-debug mixed mode)
:
----- THREAD ------
Current thread (0x001e9350): JavaThread "CompilerThread1" daemon
[ thread in vm, id=20]
Stack: [0xb2500000,0xb2580000), sp=0xb257e500, free space=505k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native
code)
V [libjvm.so+0xc3b13c]
:
Current CompileTask:
opto: 11 java.lang.Thread.setPriority(I)V (53 bytes)
----- P R O C E S S ------
Java Threads: ( => current thread )
  0x00229930 JavaThread "Low Memory Detector" daemon [ thread blocked, id=21]
=>0x001e9350 JavaThread "CompilerThread1" daemon [ thread in vm, id=20]
 :
```

In this case, there are two potential workarounds:

- The brute force approach: Change the configuration so that the application is run with the -client option to specify the HotSpot Client VM.
- The subtle approach: Assume that the bug only occurs during the compilation of the java.lang.Thread.setPriority method, and exclude this method from compilation.

The first approach (to use the -client option) might be trivial to configure in some environments. In others, it might be more difficult if the configuration is complex or if the command line to configure the VM is not readily accessible. In general, switching from the HotSpot Server VM to the HotSpot Client VM also reduces the peak performance of an application. Depending on the environment, this might be acceptable until the issue is diagnosed and fixed.

The second approach (exclude the method from compilation) requires creating the file .hotspot\_compiler in the working directory of the application. The following example shows this approach.

exclude java/lang/Thread setPriority

In general, the format of this file is excludeclassmethod, where class is the class (fully qualified with the package name) and method is the name of the method. Constructor methods are specified as <init> and static initializers are specified as <clinit>.



#### Note:

The.hotspot\_compiler file is an unsupported interface. It is documented here solely for the purposes of troubleshooting and finding a temporary workaround.

After the application is restarted, the compiler will not attempt to compile any of the methods excluded in the .hotspot\_compiler file. In some cases this can provide temporary relief until the root cause of the crash is diagnosed and the bug is fixed.

In order to verify that the HotSpot VM correctly located and processed the .hotspot\_compiler file that is shown in the previous example from the second approach, look for the log information at runtime.



### Working Around Crashes During Garbage Collection

Possible workaround if the crash occurs during garbage collection.

If a crash occurs during garbage collection (GC), then the fatal error log reports that a VM\_Operation is in progress. For the purpose of this discussion, assume that the mostly concurrent GC (-XX:+UseConcMarkSweep) is not in use. The VM\_Operation is shown in the THREAD section of the log and indicates one of the following situations:

- Generation collection for allocation
- Full generation collection
- Parallel GC failed allocation
- Parallel GC failed permanent allocation
- Parallel GC system GC

Most likely, the current thread reported in the log is the VMThread. This is the special thread used to execute special tasks in the HotSpot VM. The following example is a fragment of the fatal error log from a crash in the serial garbage collector.

----- T H R E A D -----Current thread (0x002cb720): VMThread [id=3252] siginfo: ExceptionCode=0xc0000005, reading address 0x00000000 Registers: EAX=0x0000000a, EBX=0x00000001, ECX=0x00289530, EDX=0x00000000 ESP=0x02aefc2c, EBP=0x02aefc44, ESI=0x00289530, EDI=0x00289530 EIP=0x0806d17a, EFLAGS=0x00010246 Top of Stack: (sp=0x02aefc2c)



```
0x02aefc2c: 00289530 081641e8 00000001 0806e4b8
0x02aefc3c: 00000001 00000000 02aefc9c 0806e4c5
0x02aefc4c: 081641e8 081641c8 00000001 00289530
0x02aefc5c: 0000000 0000000 0000001 0000001
0x02aefc6c: 0000000 0000000 0000000 08072a9e
0x02aefc7c: 00000000 00000000 0000000 00035378
0x02aefc8c: 00035378 00280d88 00280d88 147fee00
0x02aefc9c: 02aefce8 0806e0f5 00000001 00289530
Instructions: (pc=0x0806d17a)
0x0806d16a: 15 08 83 3d c0 be 15 08 05 53 56 57 8b f1 75 0f
0x0806d17a: Of be 05 00 00 00 00 83 c0 05 a3 c0 be 15 08 8b
Stack: [0x02ab0000,0x02af0000), sp=0x02aefc2c, free space=255k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native
code)
V [jvm.dll+0x6d17a]
  [jvm.dll+0x6e4c5]
V
V [jvm.dll+0x6e0f5]
V [jvm.dll+0x71771]
V [jvm.dll+0xfd1d3]
V
  [jvm.dll+0x6cd99]
V [jvm.dll+0x504bf]
V [jvm.dll+0x6cf4b]
V [jvm.dll+0x1175d5]
V
  [jvm.dll+0x1170a0]
V [jvm.dll+0x11728f]
V [jvm.dll+0x116fd5]
C [MSVCRT.dll+0x27fb8]
C [kernel32.dll+0x1d33b]
```

```
VM_Operation (0x0373f71c): generation collection for allocation, mode: safepoint, requested by thread 0x02db7108
```

#### Note:

A crash during garbage collection does not suggest a bug in the garbage collection implementation. It could also indicate a compiler or runtime bug, or some other issue.

You can try the following workarounds if you repeatedly get a crash during garbage collection:

- Switch GC configuration. For example, if you are using the serial collector, then try the throughput collector, or vice versa.
- If you are using the HotSpot Server VM, then try the HotSpot Client VM.

If you are not sure which garbage collector is in use, then you can use the jmap utility on the Oracle Solaris and Linux operating systems. See The jmap Utility to get the heap information from the core file, if the core file is available. In general, if the GC configuration is not specified on the command line, then the serial collector will be used on Windows. On the Oracle Solaris and Linux operating systems, it depends on the machine configuration. If the machine has at least 2 GB of memory and has at least 2 CPUs, then the throughput collector (Parallel GC) will be used. For smaller machines, the serial collector is the default. The option



to select the serial collector is -XX:+UseSerialGC and the option to select the throughput collector is -XX:+UseParallelGC. If, as a workaround, you switch from the throughput collector to the serial collector, then you might experience some performance degradation on multiprocessor systems. This might be acceptable until the root issue is diagnosed and fixed.

## Working Around Crashes Caused by Class Data Sharing

When the JRE is installed, the installer loads a set of classes from the system JAR file into a private internal representation and dumps that representation to a file called a shared archive. When the JVM starts, the shared archive is memory-mapped to allow sharing of read-only JVM metadata for these classes among multiple JVM processes. The startup time is reduced thus saving the cost because restoring the shared archive is faster than loading the classes. Class data sharing is supported with the Java HotSpot VM. The G1, serial, parallel, and parallelOldGC garbage collectors are supported. The shared string feature (part of class data sharing) supports only the G1 garbage collector on non-Windows platforms.

The fatal error log prints the version string in the header of the log. If sharing is enabled, it is indicated by the text "sharing," as shown in the following example.

```
# An unexpected error has been detected by HotSpot Virtual Machine:
#
# EXCEPTION_ACCESS_VIOLATION (0xc0000005) at pc=0x08083d77, pid=3572,
tid=784
#
# Java VM: Java HotSpot(TM) Client VM (1.5-internal mixed mode,
sharing)
# Problematic frame:
# V [jvm.dll+0x83d77]
```

CDS can be disabled by providing the -Xshare:off option on the command line. If the crash only occurs with sharing enabled, then it is possible that you encountered a bug in this feature. In that case, gather as much information as possible and submit a bug report.

## Microsoft Visual C++ Version Considerations

If you experience a crash with a Java application and if you have native or JNI libraries that are compiled with a different release of the compiler, then you must consider compatibility issues between the runtimes. Specifically, your environment is supported only if you follow the Microsoft guidelines when dealing with multiple runtimes. For example, if you allocate memory using one runtime, then you must release it using the same runtime. Unpredictable behavior or crashes can happen if you release a resource using a different library than the one that allocated the resource.



#### Note:

Use the java command option -Xinternalversion to determine which version of Microsoft Visual Studio built the JDK. This version may vary depending on the JDK release.



# Troubleshoot Process Hangs and Loops

This chapter provides information and guidance about some specific procedures for troubleshooting hanging or looping processes.

Problems can occur that involve hanging or looping processes. A hang can occur for many reasons, but often stems from a deadlock in an application code, API code, or library code. A hang can be due to a bug in the Java HotSpot VM.

Sometimes an apparent hang turns out to be, in fact, a loop. For example, a bug in a VM process that causes one or more threads to go into an infinite loop can consume all available CPU cycles.

The initial step when you diagnose a hang is to find out if the VM process is idle or consuming all available CPU cycles. You can do this using a native operating system (OS) utility. If the process appears to be busy and is consuming all available CPU cycles, then it is likely that the issue is a looping thread rather than a deadlock. On the Oracle Solaris operating system, for example, the command prstat -L -p pid can be used to report the statistics for all lightweight processes (LWPs) in the target process and therefore will identify the threads that are consuming a lot of CPU cycles.

This chapter contains the following sections:

- Diagnose a Loop Process
- Diagnose a Hung Process
- Oracle Solaris 8 Thread Library

## Diagnose a Loop Process

If a VM process appears to be looping, try to get a thread dump. A thread dump often makes it clear which thread is looping, and the trace stack in the thread dump can provide the direction on where (and maybe why) the thread is looping.

If the application console (standard input/output) is available, then press the Control+\ key combination (on Oracle Solaris or Linux) or the Control+Break key combination (on Windows) to cause the HotSpot VM to print a thread dump, including thread state. On Oracle Solaris and Linux operating systems the thread dump can also be obtained by sending a SIGQUIT to the process (command kill -QUIT pid). In this case, the thread dump is printed to the standard output of the target process. The output might be directed to a file, depending on how the process was started.

If the Java process is started with the -XX:+PrintClassHistogram command-line option, then the Control+Break handler will produce a heap histogram.

If a thread dump can be obtained, then a good place to start is the thread stacks of the threads that are in the RUNNABLE state. See Thread Dump, for more information about the format of the thread dump, as well as a table of the possible thread states in the thread dump. In some cases, it might be necessary to get a sequence of thread dumps in order to determine which threads appear to be continuously busy.



If the application console is not available (for example, the process is running in the background, or the VM output is directed to an unknown location), then the jstack utility or the jhsdb jstack utility can be used to get the stack thread. See The jstack Utility or the jstack mode of jhsdb for more about the output of these utilities. The jstack utility or the jhsdb jstack utility should also be used if the thread dump does not provide any evidence that a Java thread is looping.

When reviewing the output of the jstack utility, focus initially on the threads that are in the RUNNABLE state. This is the most likely state for threads that are busy and possibly looping. It might be necessary to execute jstack a number of times to get a better idea of which threads are looping. If a thread appears to be always in the RUNNABLE state, then use jhsdb jstack with the --mixed option to print the native frames and provide a further hint about what the thread is doing. If a thread appears to be looping continuously while in the RUNNABLE state, then this situation can indicate a potential HotSpot VM bug that needs further investigation.

If the VM does not respond to Control+\, then this could indicate a VM bug rather than an issue with the application or library code. In this case, use jhsdb jstack with the --mixed option to get a thread stack for all threads. The output will include the thread stacks for VM internal threads. In this stack trace, identify threads that do not appear to be waiting. For example, on the Oracle Solaris operating system, you identify the threads that are not in functions such as \_\_lwp\_cond\_wait, \_\_lwp\_park, \_\_\_pollsys, or other blocking functions. If it appears that the looping is caused by a VM bug, then collect as much data as possible and submit a bug report. See Submit a Bug Report for more about data collection.

## Diagnose a Hung Process

Use the thread dump to diagnose a hung process.

If the application appears to be hung and the process appears to be idle, then the first step is to try to get a thread dump. If the application console is available, then press Control+\ (on Oracle Solaris or Linux), or Control+Break (on Windows) to cause the HotSpot VM to print a thread dump. On the Oracle Solaris and Linux operating systems, the thread dump can also be obtained by sending a SIGQUIT to the process (command kill -QUIT pid). If the hung process can generate a thread dump, then the output is printed to the standard output of the target process.

After printing the thread dump, the HotSpot VM executes a deadlock detection algorithm.

The following sections describe various situations for a hung process.

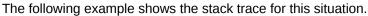
- Deadlock Detected
- Deadlock Not Detected
- No Thread Dump

#### **Deadlock Detected**

If a deadlock is detected, then it will be printed along with the stack trace of the threads involved in the deadlock.



Found one Java-level deadlock: "AWT-EventQueue-0": waiting to lock monitor 0x000ffbf8 (object 0xf0c30560, a java.awt.Component\$AWTTreeLock), which is held by "main" "main": waiting to lock monitor 0x000ffe38 (object 0xf0c41ec8, a java.util.Vector), which is held by "AWT-EventQueue-0" Java stack information for the threads listed above: \_\_\_\_\_ "AWT-EventQueue-0": at java.awt.Container.removeNotify(Container.java:2503) - waiting to lock <0xf0c30560> (a java.awt.Component\$AWTTreeLock) at java.awt.Window\$1DisposeAction.run(Window.java:604) at java.awt.Window.doDispose(Window.java:617) at java.awt.Dialog.doDispose(Dialog.java:625) at java.awt.Window.dispose(Window.java:574) at java.awt.Window.disposeImpl(Window.java:584) at java.awt.Window\$1DisposeAction.run(Window.java:598) - locked <0xf0c41ec8> (a java.util.Vector) at java.awt.Window.doDispose(Window.java:617) at java.awt.Window.dispose(Window.java:574) at javax.swing.SwingUtilities\$SharedOwnerFrame.dispose(SwingUtilities.java:1743) at. javax.swing.SwingUtilities\$SharedOwnerFrame.windowClosed(SwingUtilities.java: 1722) at java.awt.Window.processWindowEvent(Window.java:1173) at javax.swing.JDialog.processWindowEvent(JDialog.java:407) at java.awt.Window.processEvent(Window.java:1128) at java.awt.Component.dispatchEventImpl(Component.java:3922) at java.awt.Container.dispatchEventImpl(Container.java:2009) at java.awt.Window.dispatchEventImpl(Window.java:1746) at java.awt.Component.dispatchEvent(Component.java:3770) at java.awt.EventQueue.dispatchEvent(EventQueue.java:463) at. java.awt.EventDispatchThread.pumpOneEventForHierarchy(EventDispatchThread.jav a:214) at java.awt.EventDispatchThread.pumpEventsForHierarchy(EventDispatchThread.java: 163) at. java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:157) at. java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:149) at java.awt.EventDispatchThread.run(EventDispatchThread.java:110) "main": at java.awt.Window.getOwnedWindows(Window.java:844) - waiting to lock <0xf0c41ec8> (a java.util.Vector) javax.swing.SwingUtilities\$SharedOwnerFrame.installListeners(SwingUtilities.j





```
ava:1697)
    at
javax.swing.SwingUtilities$SharedOwnerFrame.addNotify(SwingUtilities.ja
va:1690)
    at java.awt.Dialog.addNotify(Dialog.java:370)
        - locked <0xf0c30560> (a java.awt.Component$AWTTreeLock)
        at java.awt.Dialog.conditionalShow(Dialog.java:441)
        - locked <0xf0c30560> (a java.awt.Component$AWTTreeLock)
        at java.awt.Dialog.show(Dialog.java:499)
        at java.awt.Component.show(Component.java:1287)
        at java.awt.Component.setVisible(Component.java:1242)
        at test01.main(test01.java:10)
```

```
Found 1 deadlock.
```

The default deadlock detection works with locks that are obtained using the synchronized keyword, as well as with locks that are obtained using the java.util.concurrent package. If the Java VM flag – XX:+PrintConcurrentLocks is set, then the stack trace also shows a list of lock owners.

If a deadlock is detected, then you must examine the output in more detail in order to understand the deadlock. In the previous example, the thread main is locking object 0xf0c30560 and is waiting to enter 0xf0c41ec8, which is locked by thread AWT-EventQueue-0. However, thread AWT-EventQueue-0 is waiting to enter 0xf0c30560, which is locked by main.

The detail in the stack traces provides information to help you find the deadlock.

### **Deadlock Not Detected**

If the thread dump is printed and no deadlocks are found, then the issue might be a bug in which a thread is waiting for a monitor that is never notified. This could be a timing issue or a general logic bug.

To find out more about the issue, examine each of the threads in the thread dump and each thread that is blocked in <code>Object.wait()</code>. The caller frame in the stack trace indicates the class and method that is invoking the <code>wait()</code> method. If the code was compiled with line number information (the default), then this provides a direction as to the code to examine. In most cases, you must have some knowledge of the application logic or library in order to diagnose this issue further. In general, you must understand how the synchronization works in the application and the details and conditions for when and where the monitors are notified.

### No Thread Dump

If the VM is deadlocked or hung, use the jstack or jhsdb jstack command.

If the VM does not respond to Control+\ or Control+Break, then it is possible that the VM is deadlocked or hung for some other reason. In that case, use The jstack Utility or jhsdb jstack to get a thread dump. This also applies in the case when the application is not accessible, or the output is directed to an unknown location.

In the thread dump, examine each of the threads in the BLOCKED state. The top frame can sometimes indicate why the thread is blocked (for example, Object.wait or



Thread.sleep). The rest of the stack will give an indication of what the thread is doing. This is particularly true when the source is compiled with line number information (the default), and you can cross-reference the source code.

If a thread is in the BLOCKED state and the reason is not clear, then use jhsdb jstack -mixed to get a mixed stack. With the mixed stack output, it should be possible to identify why the thread is blocked. If a thread is blocked trying to enter a synchronized method or block, then you will see frames such as ObjectMonitor::enter near the top of the stack. The following example shows a sample, mixed-stack output.

```
------ t@13 -----

0xff31e8b8 __lwp_cond_wait + 0x4

0xfea8c810 void ObjectMonitor::EnterI(Thread*) + 0x2b8

0xfeac86b8 void ObjectMonitor::enter2(Thread*) + 0x250

:
```

Threads in the RUNNABLE state might also be blocked. The top frames in the mixed stack should indicate what the thread is doing.

One specific thread to check is VMThread. This is the special thread used to execute operations like garbage collection (GC). It can be identified as the thread that is executing VMThread::run() in its initial frames. On the Oracle Solaris, it is typically t@4. On Linux, it should be identifiable using the C++ mangled name ZN8VMThread4loopEv.

In general, the VM thread is in one of three states: waiting to execute a VM operation, synchronizing all threads in preparation for a VM operation, or executing a VM operation. If you suspect that a hang is a HotSpot VM bug rather than an application or class library deadlock, then pay special attention to the VM thread.

If the VM thread appears to be stuck in SafepointSynchronize::begin, then this could indicate an issue bringing the VM to a safepoint. A safepoint indicates that all threads executing in the VM are blocked and waiting for a special operation, such as GC, to complete.

If the VM thread appears to be stuck in function, where function ends in doit, then this could also indicate a VM problem.

In general, if you can execute the application from the command line, and you get to a state where the VM does not respond to Control+\ or Control+Break, it is more likely that you have uncovered a VM bug, a thread library issue, or a bug in another library. When this occurs, get a crash dump. See Collect Core Dumps for instructions about gathering as much information as possible, and submit a bug report or call support.

One other tool to mention in the context of hung processes is the pstack utility on the Oracle Solaris operating system. On the Oracle Solaris 8 and 9 operating systems, this utility prints the thread stacks for LWPs in the target process. On the Oracle Solaris 10 operating system and starting with the JDK 5.0 release, the output of pstack is similar, though not identical, to the output from jhsdb jstack --mixed.The Oracle Solaris 10 operating system implementation of pstack prints the fully qualified class name, method name, and bytecode index (BCI). It will also print line numbers for cases where the source was compiled with line number information (the default). This is useful for developers and administrators who are familiar with the other utilities on the Oracle Solaris operating system that exercise features of the /proc file system.



The equivalent tool of pstack on Linux is lsstack. This utility is included in some distributions and otherwise obtained from sourceforge. At the time of this writing, lsstack reported native frames only.

# **Oracle Solaris 8 Thread Library**

The default thread library on the Oracle Solaris 8 operating system is often referred to as the T1 library. This thread library implemented the *m*:*n* threading model, where *m* user threads are mapped to *n* kernel-level threads (LWPs). The Oracle Solaris 8 operating system also shipped with an alternative and newer thread library in /usr/lib/lwp. The alternative thread library is often referred to as the T2 library, and it became the default thread library in the Oracle Solaris 9 and 10 operating systems. In older releases of J2SE (pre-1.4.0 in particular), there were a number of issues with the default thread library (for example, bugs in the thread library, LWP synchronization problems, or LWP starvation). LWP starvation is a scenario in which there are user threads in the RUNNABLE state, but there are no kernel level threads available.

Although the issues cited are historical, it should be noted that when the JDK software is deployed on the Oracle Solaris 8 operating system, it still uses the T1 library by default. LWP starvation type issues do not happen because the JDK release uses "bound threads" so that each user thread is bound to a kernel thread. However, in the event that you encounter an issue, such as a hang, that you believe is a thread library issue, then you can instruct the HotSpot VM to use the T2 library by adding /usr/lib/lwp to the LD\_LIBRARY\_PATH. To check if the T2 library is in use, issue the command pldd *pid* to list the libraries loaded by the specified process.



# 7 Handle Signals and Exceptions

This chapter provides information about how signals and exceptions are handled by the Java HotSpot Virtual Machine. It also describes the signal chaining facility, available on the Oracle Solaris, Linux, and macOS operating systems, which facilitates writing applications that must install their own signal handlers.

This chapter contains the following sections:

- Handle Signals on Oracle Solaris, Linux, and macOS
- Handle Exceptions on Windows
- Signal Chaining
- Handle Exceptions Using the Java HotSpot VM
- Console Handlers
- Signals Used in Oracle Solaris, Linux, and macOS

# Handle Signals on Oracle Solaris, Linux, and macOS

The Java HotSpot VM installs signal handlers to implement various features and to handle fatal error conditions.

For example, in an optimization to avoid explicit null checks in cases where java.lang.NullPointerException will be thrown rarely, the SIGSEGV signal is caught and handled, and the NullPointerException is thrown.

In general, there are two categories where signal/traps happen:

- When signals are expected and handled, like implicit null-handling. Another example is
  the safepoint polling mechanism, which protects a page in memory when a safepoint is
  required. Any thread that accesses that page causes a SIGSEGV, which results in the
  execution of a stub that brings the thread to a safepoint.
- Unexpected signals. This includes a SIGSEGV when executing in VM code, Java Native Interface (JNI) code, or native code. In these cases, the signal is unexpected, so fatal error handling is invoked to create the error log and terminate the process.

Table 7-2 lists the signals that are currently used on the Oracle Solaris, Linux, and macOS operating systems.

## Handle Exceptions on Windows

On Windows, an exception is an event that occurs during the execution of a program.

There are two kinds of exceptions: hardware exceptions and software exceptions. Hardware exceptions are comparable to signals such as SIGSEGV and SIGKILL on the Oracle Solaris and Linux operating systems. Software exceptions are initiated explicitly by applications or the operating system using the RaiseException() API.



On Windows, the mechanism for handling both hardware and software exceptions is called structured exception handling (SEH). This is stack frame-based exception handling similar to the C++ and Java exception handling mechanism. In C++, the \_\_try and \_\_except keywords are used to guard a section of code that might result in an exception, as shown in the following example.

```
__try {
    // guarded body of code
} __except (filter-expression) {
    // exception-handler block
}
```

The \_\_except block is filtered by a filter expression that uses the integer exception code returned by the GetExceptionCode() API, exception information returned by the GetExceptionInformation() API, or both.

The filter expression should evaluate to one of the following values:

• EXCEPTION CONTINUE EXECUTION = -1

The filter expression repaired the situation, and execution continues where the exception occurred. Unlike some exception schemes, SEH supports the **resumption model** as well. This is much like the UNIX signal handling in the sense that after the signal handler finishes, the execution continues where the program was interrupted. The difference is that the handler in this case is just the filter expression itself and not the \_\_except block. However, the filter expression might also involve a function call.

• EXCEPTION CONTINUE SEARCH = 0

The current handler cannot handle this exception. Continue the handler search for the next handler. This is similar to the <code>catch</code> block not matching an exception type in C++ and Java.

• EXCEPTION EXECUTE HANDLER = 1

The current handler matches and can handle the exception. The  $\_\__{except}$  block is executed.

The <u>try</u> and <u>finally</u> keywords are used to construct a termination handler, as shown in the following example.

```
__try {
    // guarded body of code
} __finally {
    // __finally block
}
```

When control leaves the \_\_try block (after an exception or without an exception), the \_\_finally block is executed. Inside the \_\_finally block, the AbnormalTermination() API can be called to test whether control continued after the exception or not.

Windows programs can also install a top-level **unhandled exception filter** function to catch exceptions that are not handled in the \_\_try/\_except block. This function is installed on a process-wide basis using the SetUnhandledExceptionFilter() API. If there is no handler for an exception, then UnhandledExceptionFilter() is



called, and this will call the top-level unhandled exception filter function, if any, to catch that exception. This function also shows a message box to notify the user about the unhandled exception.

Windows exceptions are comparable to Unix synchronous signals that are attributable to the current execution stream. In Windows, asynchronous events such as console events (for example, the user pressing Control+C at the console) are handled by the console control handler registered using the SetConsoleCtlHandler() API.

If an application uses the signal() API on Windows, then the C runtime library (CRT) maps both Windows exceptions and console events to appropriate signals or C runtime errors. For example, CRT maps Control+C to SIGINT and all other console events to SIGBREAK. Similarly, if you register the SIGSEGV handler, CRT translates the corresponding exception to a signal. CRT startup code implements a \_\_try/\_except block around the main() function. The CRT's exception filter function (named \_XcptFilter) maps the Win32 exceptions to signals and dispatches signals to their appropriate handlers. If a signal's handler is set to SIG\_DFL (default handling), then XcptFilter calls UnhandledExceptionFilter.

The **vectored exception handling** mechanism can also be used. Vectored handlers are not frame-based handlers. A program can register zero or more vectored exception handlers using the AddVectoredExceptionHandler API. Vectored handlers are invoked before structured exception handlers, if any, are invoked, regardless of where the exception occurred.

vectored exception handler returns one of the following values:

- EXCEPTION CONTINUE EXECUTION: Skip the next vectored and SEH handlers.
- EXCEPTION CONTINUE SEARCH: Continue to the next vectored or SEH handler.

See the Microsoft website to know more on Windows exception handling.

# Signal Chaining

Signal chaining enables you to write applications that need to install their own signal handlers. This facility is available on Solaris, Linux, and macOS.

The signal chaining facility has the following features:

 Support for preinstalled signal handlers when you create Oracle's HotSpot Virtual Machine.

When the HotSpot VM is created, the signal handlers for signals that are used by the HotSpot VM are saved. During execution, when any of these signals are raised and are not to be targeted at the HotSpot VM, the preinstalled handlers are invoked. In other words, preinstalled signal handlers are *chained* behind the HotSpot VM handlers for these signals.

• Support for the signal handlers that are installed after you create the HotSpot VM, either inside the Java Native Interface code or from another native thread.

Your application can link and load the libjsig.so shared library before the libc/ libthread/libpthread library. This library ensures that calls such as signal(), sigset(), and sigaction() are intercepted and don't replace the signal handlers that are used by the HotSpot VM, if the handlers conflict with the signal handlers that are already installed by HotSpot VM. Instead, these calls save the new signal handlers. The new signal handlers are chained behind the HotSpot VM signal handlers for the signals.



During execution, when any of these signals are raised and are not targeted at the HotSpot VM, the preinstalled handlers are invoked.

If support for signal handler installation after the creation of the VM is not required, then the libjsig.so shared library is not needed.

To enable signal chaining, perform one of the following procedures to use the libjsig.so shared library:

 Link the libjsig.so shared library with the application that creates or embeds the HotSpot VM:

```
cc -L libjvm.so-directory -ljsig -ljvm java application.c
```

- Use the LD PRELOAD environment variable:
  - \* Korn shell (ksh):

```
export LD_PRELOAD=libjvm.so-directory/libjsig.so;
java_application
```

\* C shell (csh):

```
setenv LD_PRELOAD libjvm.so-directory/libjsig.so;
java application
```

The interposed signal(), sigset(), and sigaction() calls return the saved signal handlers, not the signal handlers installed by the HotSpot VM and are seen by the operating system.

#### Note:

The SIGQUIT, SIGTERM, SIGINT, and SIGHUP signals cannot be chained. If the application must handle these signals, then consider using the –Xrs option.

#### **Enable Signal Chaining in macOS**

To enable signal chaining in macOS, set the following environment variables:

- DYLD\_INSERT\_LIBRARIES: Preloads the specified libraries instead of the LD PRELOAD environment variable available on Solaris and Linux.
- DYLD\_FORCE\_FLAT\_NAMESPACE: Enables functions in the libjsig library and replaces the OS implementations, because of macOS's two-level namespace (a symbol's fully qualified name includes its library). To enable this feature, set this environment variable to any value.

The following command enables signal chaining by preloading the libjsig library:

```
$ DYLD_FORCE_FLAT_NAMESPACE=0 DYLD_INSERT_LIBRARIES="JAVA_HOME/lib/
libjsig.dylib" java MySpiffyJavaApp
```



#### Note:

The library file name on macOS is libjsig.dylib not libjsig.so as it is on Solaris or Linux.

# Handle Exceptions Using the Java HotSpot VM

The HotSpot VM installs a top-level exception handler during initialization using the AddVectoredExceptionHandlerAPI for 64-bit systems.

It also installs the Win32 SEH using a  $\_try/\_except$  block in C++ around the thread (internal) start function call for each thread created.

Finally, it installs an exception handler around JNI functions.

If an application must handle structured exceptions in JNI code, then it can use \_\_try / \_\_except statements in C++. However, if it must use the vectored exception handler in JNI code, then the handler must return EXCEPTION\_CONTINUE\_SEARCH to continue to the VM's exception handler.

In general, there are two categories in which exceptions happen:

- When exceptions are expected and handled. Examples include the implicit null handling cited, previously where accessing a null causes an EXCEPTION\_ACCESS\_VIOLATION, which is handled.
- Unexpected exceptions. An example is an EXCEPTION\_ACCESS\_VIOLATION when executing in VM code, in JNI code, or in native code. In these cases, the signal is unexpected, and fatal error handling is invoked to create the error log and terminate the process.

## **Console Handlers**

This topic describes a list of console events that are registered with the Java HotSpot VM.

The Java HotSpot VM registers console events, as shown in Table 7-1.

Console Event	Signal	Usage
CTRL_C_EVENT	SIGINT	This event and signal is used to terminate a process. (Optional)
CTRL_CLOSE_EVENTCTRL_LO GOFF_EVENTCTRL_SHUTDOWN _EVENT	SIGTERM	This event and signal is used by the shutdown hook mechanism when the VM is terminated abnormally. (Optional)
CTRL_BREAK_EVENT	SIGBREAK	This event and signal is used to dump Java stack traces at the standard error stream. (Optional)

Table 7-1 Console Events

If an application must register its own console handler, then the -Xrs option can be used. With this option, shutdown hooks are not run on SIGTERM (with the previously shown mapping of events), and thread dump support is not available on SIGBREAK (with the above mapping of the Control+Break event).



# Signals Used in Oracle Solaris, Linux, and macOS

This topic describes a list of signals that are used on Solaris OS, Linux, and macOS

Signal	Description
SIGSEGV, SIGBUS, SIGFPE, SIGPIPE, SIGILL	These signals are used in the implementation for implicit null check, and so forth.
SIGQUIT	This signal is used to dump Java stack traces to the standard error stream. (Optional)
SIGTERM, SIGINT, SIGHUP	These signals are used to support the shutdown hook mechanism (java.lang.Runtime.addShutdownHook) when the VM is terminated abnormally. (Optional)
SIGJVM1, SIGJVM2	These signals are reserved for use by the Java Virtual Machine. (Solaris only)
SIGUSR2	This signal is used internally on Linux and macOS. It is not used by the VM on Solaris.
SIGABRT	The HotSpot VM does not handle this signal. Instead, it calls the abort function after fatal error handling. If an application uses this signal, then it should terminate the process to preserve the expected semantics.

 Table 7-2
 Signals Used on Oracle Solaris, Linux, and macOS

Signals tagged as "optional" are not used when the -Xrs option is specified to reduce signal usage. With this option, fewer signals are used, although the VM installs its own signal handler for essential signals such as SIGSEGV. Specifying this option means that the shutdown hook mechanism will not execute if the process receives a SIGQUIT, SIGTERM, SIGINT, or SIGHUP. Shutdown hooks will execute, as expected, if the VM terminates normally (that is, when the last non-daemon thread completes or the System.exit method is invoked).

SIGUSR2 is used to implement, suspend, and resume on Linux and macOS. However, it is possible to specify an alternative signal to be used instead of SIGUSR2. This is done by specifying the \_JAVA\_SR\_SIGNUM environment variable. If this environment variable is set, then it must be set to a value larger than the maximum of SIGSEGV and SIGBUS.

# Part III Debug Core Library Issues

This part describes issues and troubleshooting techniques that arise with time zone settings and contains the following topic.

• Time Zone Settings in the JRE

Describes some issues that arise with time zone settings with Java Runtime Environment (JRE) and troubleshooting techniques to resolve these issues.



# 8 Time Zone Settings in the JRE

This chapter describes some issues that can arise with time zone settings with the Java Runtime Environment (JRE) on the Windows operating system. It further describes troubleshooting techniques and workarounds to solve these issues. This chapter contains the following sections:

- Native Time Zone Information and the JRE
- Determine the Default Time Zone on Windows

# Native Time Zone Information and the JRE

The JRE reads the native time zone information to determine your default time zone.

For example, on Windows, the JRE queries the registry to determine the default time zone.

However, the JRE also maintains its own time zone database. This provides cross-platform support because the different operating system APIs are not sufficient to support the Java APIs. The Java time zone database supports time zone IDs and determines daylight saving time rules for all the time zones that the JRE supports. The tzupdater tool is available for download from the Java SE Download Page.

Modifications to the JRE for each specific operating system are necessary so that the operating system can deliver the system time to the JRE. Then, if a Java application requests the system date by calling date and time related constructors, the system time is returned.

Examples of such constructors are:

```
java.util.Date()
java.util.GregorianCalendar()
```

Constructors related to date and time include:

```
System.currentTimeMillis()
System.nanoTime()
```

Operating system-specific patches might be required to ensure that the correct system time is delivered to the JRE.

The following sections describe troubleshooting techniques for time zone settings.

- Determine the Time Zone Data Version in Use
- Troubleshoot Problems with TZupdater

### Determine the Time Zone Data Version in Use

The time zone database version that ships in any Java runtime from Oracle is documented in the release notes. However, the actual version can be different from the version mentioned



there if the Java runtime was patched using the Java time zone updater tool called tzupdater.

To determine the current time zone data version of your Java runtime using the tzupdater tool, run the tool with the -V option as shown in the following example:

```
java -jar tzupdater.jar -V
```

Here is a typical output from running the tzupdater tool.

```
tzupdater version 2.2.0-b01
JRE tzdata version: tzdata2018g
```

You can download the tzupdater tool from this web page: Timezone Updater Tool.

### Troubleshoot Problems with TZupdater

Sometimes, when you run tzupdater, it quits with the message: "There's no tzdata available for this Java runtime." The following are two examples.

```
$ java -jar tzupdater.jar -V
tzupdater version 2.1.1-b01
JRE tzdata version: tzdata2017b
There's no tzdata available for this Java runtime.
```

The likely cause is that you are using a Java runtime that is not from Oracle. Oracle provides the Java runtime for Oracle Solaris (x64, SPARC), Linux (x64), Microsoft Windows (x64), and macOS (x64). The java.vendor property value for these is Sun Microsystems Inc., Oracle Corporation, or BEA Systems, Inc. Oracle does not provide the Java runtime for other platforms.

The output of running the java -version command does not provide enough information to determine the actual vendor of a Java runtime. However, running tzupdater in update mode with the -v option does print out the java.vendor property. The following example shows the result of running tzupdater when the environment is HP\_UX from Hewlett Packard.

```
root@my_server:/opt/java6/bin> uname -a
HP-UX my_server B.11.23 U ia64 1114591084 unlimited-user license
root@my_server:/opt/java6/bin> ./java -version
java version "1.6.0.05"
Java(TM) SE Runtime Environment (build 1.6.0.05-
jinteg_14_oct_2009_01_44-b00)
Java HotSpot(TM) Server VM (build 14.2-b01-jre1.6.0.05-rc5, mixed mode)
root@my_server:/opt/java6/bin> ./java -jar tzupdater.jar -v -l
java.home: /opt/java6/jre
java.vendor: Hewlett-Packard Co.
java.version: 1.6.0.05
JRE tzdata version: tzdata2009i
There's no tzdata available for this Java runtime.
```

In the previous example, java.vendor is set to "Hewlett-Packard Co." The Java runtime that you are trying to update using tzupdater is not supported by Oracle.



A possible solution is to visit the website of your Java runtime vendor and determine whether a time zone updater tool is available.

# Determine the Default Time Zone on Windows

This section clarifies how the JRE determines the default time zone on the Windows 10 and later operating systems. If the expected time zone isn't reported, then use the troubleshooting techniques provided in the following sections:

- Check the Default Time Zone Java Runtime Reports
- Determine the Setting in the Control Panel
- Check for Automatic Daylight Saving Time Adjustment
- Set the Default Time Zone in Windows Settings
- Check -Duser.timezone System Property
- Special Tool in Windows
- Internal Representation of Time Zone Mappings

## Check the Default Time Zone Java Runtime Reports

You can write a simple program to determine which time zone the JDK reports the default time zone-based on a check with the native operating system.

The Java program in the following example returns the default time zone:

```
public class DefaultTimeZone {
    public static void main(String[] args) {
        System.out.println(java.util.TimeZone.getDefault().getID());
    }
}
```

You can save the code snippet in the previous example to a file named DefaultTimeZone.java and compile it using the javac command. Then, you can run the compiled DefaultTimeZone class, as shown in the following example.

```
c:\tztest> javac DefaultTimeZone.java
c:\tztest> java DefaultTimeZone
Europe/Berlin
```

In the previous example, the default time zone is Europe/Berlin. Running the program should display your local time zone. If the output is not the expected time zone, then continue with the following troubleshooting steps.

#### Determine the Setting in the Control Panel

You can change or examine the system's default time zone using Windows Settings or the Windows Control Panel. For example, you can select this time zone setting in Windows 10:

(UTC+01:00) Amsterdam, Berlin, Bern, Rome, Stockholm, Vienna

The corresponding value for the Registry key TimeZoneKeyName is "W. Europe Standard Time."



## Check for Automatic Daylight Saving Time Adjustment

You can check whether the automatic adjustment of daylight saving time is enabled through the graphical user interface (GUI) or through the Windows registry.

- GUI Method: To use the Control Panel to check whether automatic adjustment of daylight saving time is enabled:
  - 1. Click the Windows Start button and then click Control Panel.
  - 2. Click Date and Time.
  - 3. Click the Change Time Zone button.
  - 4. There is a check box labeled "Automatically adjust time for Daylight Savings Time. "See if this check box is selected, and change the setting if you want.
  - 5. Click OK. This returns you to the Date and Time dialog box.
- **Windows Registry Method:** You can run Windows Registry Editor to check whether automatic adjustment of daylight saving time is enabled.

#### Note:

It is a good practice to back up the Windows registry before reviewing or editing it. If you make a mistake, you can damage the Windows registry.

To enable the automatic adjustment of daylight saving time from the Windows registry:

- 1. Click the Windows Start button.
- 2. In the Search programs and files field, enter regedit and then press Enter to open the Registry Editor.
- 3. In the Registry Editor, search for the key DynamicDaylightTimeDisabled and look at the setting.

If the registry setting is 1, then dynamic daylight time is disabled.

If the registry setting is 0, then dynamic daylight time is enabled.

If you prefer, you can access the Windows registry from the Windows command window.

In the following example, the registry setting is 1. With this setting, the clock is not automatically adjusted for daylight saving time.

[HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Control\TimeZoneInformation] "DynamicDaylightTimeDisabled"=dword:00000001

If you disable the DynamicDaylightTimeDisabled option, then Java returns a GMT (Greenwich Mean Time) offset and not a time zone ID that is compatible with the uniform naming convention (such as "Europe/Berlin"). For example, the offset will be expressed as GMT+01 and not "Europe/Berlin."



## Set the Default Time Zone in Windows Settings

You can change or review the system's default time zone by using Windows Settings.

To set the system's default time zone from Windows Settings:

- 1. Click the Windows Start button.
- Click Settings.
- 3. Click Time & Language.
- 4. From the **Time zone** drop-down list, select your preferred time zone.

For example, you can select this time zone in Windows 10:

(UTC)+1:00) Amsterdam, Berlin, Bern, Rome, Stockholm, Vienna.

The corresponding value for the Registry key TimeZoneKeyName is "W. Europe Standard Time."

## Check -Duser.timezone System Property

You can explicitly set a default time zone on the command line by using the Java system property called user.timezone. This bypasses the settings in the Windows operating system and can be a workaround. For instance, this setting is useful if you want daylight saving time (DST) only for a single Java program running on the system.

The following example shows the system property -Duser.timezone. Compile the DefaultTimeTestZone.java program discussed in Check the Default Time Zone Java Runtime Reports from the Windows Command Prompt window. Run the following command:

```
c:\tztest> java -Duser.timezone=America/New_York DefaultTimeTestZone America/
New York
```

If setting a default time zone explicitly by specifying -Duser.timezone works for the DefaultTimeTestZone program, but does not work for your program, you should check whether your code overwrites the default Java time zone during runtime with a method call such as this:

TimeZone.setDefault(TimeZone zone)

## Special Tool in Windows

The Windows operating system provides a tool called tzutil.exe. With this tool, you can request the current time zone ID abbreviation without manually reading the registry.

Here is an example of running tzutil.exe. The first line is the command that you enter in the Windows Command Prompt window. The second line is the system response.

tzutil /g

W. Europe Standard Time



#### Internal Representation of Time Zone Mappings

On Windows, the Java runtime uses a file <java-home>\lib\tzmappings to represent the mapping between Windows and Java time zones. Each line in the file has three tokens. The first token is the Windows time zone registry key called TimeZoneKeyName. See Determine the Setting in the Control Panel.

The second token is a country code or the default code 001, which is the UN M49 code meaning "World". The third token represents the Java time zone ID.

If you select the time zone called (UTC+01:00) Amsterdam, Berlin, Bern, Rome, Stockholm, Vienna in the Windows Control Panel, then the relevant lines in the file tzmappings are:

W. Europe Standard Time: AD: Europe/Andorra: W. Europe Standard Time:AT:Europe/Vienna: W. Europe Standard Time:CH:Europe/Zurich: W. Europe Standard Time:DE:Europe/Berlin: W. Europe Standard Time:GI:Europe/Gibraltar: W. Europe Standard Time: IT: Europe/Rome: W. Europe Standard Time:LI:Europe/Vaduz: W. Europe Standard Time:LU:Europe/Luxembourg: W. Europe Standard Time:MC:Europe/Monaco: W. Europe Standard Time:MT:Europe/Malta: W. Europe Standard Time:NL:Europe/Amsterdam: W. Europe Standard Time:NO:Europe/Oslo: W. Europe Standard Time:SE:Europe/Stockholm: W. Europe Standard Time:SJ:Arctic/Longyearbyen: W. Europe Standard Time:SM:Europe/San Marino: W. Europe Standard Time:VA:Europe/Vatican: W. Europe Standard Time:001:Europe/Berlin:

In this example, the Java runtime recognizes your default time zone (token number three) based on your country. For example, if your country code is AD, then your default time zone is "Europe/Andorra".

If there is no appropriate mapping entry in the tzmappings file, then it is possible that Microsoft introduced a new time zone in a Windows update and that the new time zone is not available to the Java runtime. In this situation, you can file a bug report, and request a new entry in the tzmappings file from Oracle Java bugs website.

A similar disconnect between the operating system and the Java runtime is possible if you ran the tool tzedit.exe. This tool is provided by Microsoft, and allows users to add new time zones. The Java runtime is unlikely to have a time zone introduced into the system by this tool. Again, the solution is to file a bug to request that a new entry be added to the tzmappings file.



# Part IV Debug Client Issues

This part describes Java client issues, troubleshooting techniques, and debugging tips for client issues. The following topics are included.

Introduction to Client Issues

Provides an overview of Java client technologies, describes Java client issues, and troubleshooting tips.

• AWT

Provides guidance on specific procedures for debugging issues that occur with Java SE Abstract Windows Toolkit (AWT).

• Java 2D

Provides guidance about troubleshooting some common issues found in Java 2D API.

Swing

Provides guidance about troubleshooting some common issues found in Java SE Swing API.

Internationalization

Provides guidance about troubleshooting some issues found in Java Internationalization.

Java Sound

Describes some issues and causes that happen with Java Sound technology and suggests workarounds.



# 9 Introduction to Client Issues

This chapter explains how the different Java SE Desktop technologies interact with each other. In addition, the chapter helps you to pinpoint the technology from which you might start troubleshooting your problem and provides general troubleshooting tips. This chapter contains the following sections:

- Java SE Desktop Technologies
- General Steps to Troubleshoot an Issue
- Identify the Type of Issue
- Basic Tools
- Java Debug Wire Protocol

# Java SE Desktop Technologies

Java SE Desktop consists of several technologies used to create rich client applications.

The desktop tools and libraries provide an interface between the Java application and the core tools and libraries of the platform, as shown in Figure 9-1.

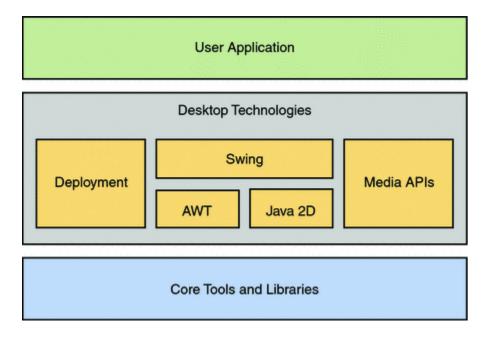


Figure 9-1 Overview of the Java SE Desktop

This topic describes troubleshooting techniques for the following Java SE desktop technologies:



 Abstract Window Toolkit (AWT) provides a set of application programming interfaces (APIs) for constructing graphical user interface (GUI) components such as menus, buttons, text fields, dialog boxes, check boxes, and for handling user input through those components. In addition, AWT allows for rendering of simple shapes such as ovals and polygons and enables developers to control the interface layout and fonts used by their applications. It also includes data transfer classes (including drag and drop) that allow cut and paste through the native platform clipboard.

The classes of this API are at the bottom of the software stack (closest to the underlying operating and desktop system).

AWT also provides a set of heavyweight components.

Purely AWT applications are usually not related to Swing. If an AWT application does custom rendering, it uses Java 2D.

• Java 2D is a set of classes for advanced 2D graphics and imaging. It encompasses line art, text, and images in a single comprehensive model. The API provides extensive support for image compositing and alpha channel images, a set of classes to provide accurate color space definition and conversion, and a rich set of display-oriented imaging operators. These classes are provided as additions to the java.awt and java.awt.image packages.

Like AWT, Java 2D is also at the bottom of the software stack (closest to the underlying operating and desktop system).

 Swing provides a comprehensive set of GUI components and services which enables the development of commercial-quality desktop and Internet/Intranet applications.

Swing is built on top of many of the other Java SE Desktop technologies, including AWT, Java2D and Internationalization. In most cases the Swing high-level components are recommended instead of those in AWT. However, there are many APIs in AWT that are important to understand when programming in Swing.

Since Swing is a lightweight toolkit, it has very little interaction with the native platform. Swing uses Java 2D for rendering, and AWT provides creation and manipulation of top-level components, such as Windows, Frames, and Dialogs.

• Internationalization is the process of designing software so that it can be adapted (localized) to various languages and regions easily, cost-effectively, and in particular without engineering changes to the software. Localization is performed by simply adding locale-specific components, such as translated text, data describing locale-specific behavior, fonts, and input methods.

In Java SE, internationalization support is fully integrated into the classes and packages that provide language-dependent or culture-dependent functionality.

To know more about internationalization APIs and features of Java SE, see Internationalization Overview.

 Java Sound provides low-level support for audio operations such as audio playback and capture (recording), mixing, musical instrument digital interface (MIDI) sequencing, and MIDI synthesis in an extensible, flexible framework. This API is supported by an efficient sound engine which guarantees high-quality audio mixing and MIDI synthesis capabilities for the platform.

The better you understand the relationships between these technologies, the more quickly you can pinpoint the area your problem falls into.

# General Steps to Troubleshoot an Issue

General steps to troubleshoot problems in your application.

When you experience problems running your application, follow the steps below for troubleshooting the issue.

#### **1**. Identify the symptom:

- Identify the Type of Issue.
- Find the problem area.
- Note the vant configuration information.

#### 2. Eliminate non-issues:

- Ensure that the correct patches, drivers, and operating systems are installed.
- Try earlier releases (back-tracing).
- Minimize the test. Restrict the test to as few issues at a time as possible.
- Minimize the hardware and software configuration. Determine if the problem is reproducible on a single system and on multiple systems. Determine if the problem changes with the browser version.
- Determine if the problem depends on whether multiple VMs are installed.

#### 3. Find the cause:

- Check for typical causes in the area.
- Use flags to change defaults.
- Use tracing.
- In exceptional cases, use system properties to temporarily change the behavior of the painting system.
- 4. Find the fix:
  - Find a possible workaround.
  - File a bug.

For guidance about how to submit a bug report and suggestions about what data to collect for the report, see Submit a Bug Report.

- Fix the setup.
- Fix the application.

# Identify the Type of Issue

Guidance about identifying the problem you are experiencing, and finding the cause and solution.

First of all, take a moment to categorize the problem you are experiencing. This will help you to identify the specific area of the problem, find the cause, and ultimately determine a solution or a workaround.

The following subsections below provide information about common issue types:



- Java Client Crashes
- Performance Problems
- Behavior Problems

Some of these might seem obvious, but it is always helpful to consider every possibility and to eliminate what is not an issue.

### Java Client Crashes

An error log is created that contains information and the state obtained at the time of the fatal error, when the Java client crashes.

The default name of the error log file is hs\_err\_*pid*.log where *pid* is the process identifier (PID) of the process that crashed. For a standalone Java application this file is created in the current directory.

To know more about the fatal error log, see Fatal Error Log.

A line near the top of the header section indicates the library where the error occurred. The following example shows that the crash was related to the AWT library.

```
# Java VM: Java HotSpot(TM) Client VM (1.6.0-beta2-b76 mixed mode,
sharing)
# Problematic frame:
# C [awt.dll+0x123456]
...
```

If the crash occurred in the Java Native Interface (JNI), it was likely to have been caused by the desktop libraries. A crash in a native library typically means a problem in Java 2D or AWT because Swing does not have much native code. The small amount of native code in Swing is then concerned with the native look and feel, and if your application is using native look and feel, then the crash may be related to this area.

The error log usually shows the exact library where the crash occurred, and this can give you a good idea of the cause. Crashes in libraries which are not part of the Java Development Kit (JDK) usually indicate problems with the environment, for example, bad video drivers or desktop managers.

### **Performance Problems**

Performance problems are harder to diagnose because you generally do not have as much information.

First, you must determine which technology has the problem. For example, rendering performance problems are probably in Java 2D, and responsiveness issues can be Swing-related.

Performance-related problems can be divided into the following categories:

Startup

How long does the application take to start up and become useful to the user?

Footprint



How much memory does the application take? This can be measured by tools such as Task Manager on Windows or top and prstat on the Oracle Solaris and Linux operating systems.

Runtime

How fast does the application complete the task it is designed to perform? For example, if the application computes something, how long does it take to finish the computations? In the case of a game, is the frame rate acceptable, and does the animation look smooth?

*Note:* This is not the same as responsiveness, which is the next topic.

Responsiveness

How fast does the application respond to user interaction? If the user clicks a menu, how long does it take for the menu to appear? Can a long-running task be interrupted? Does the application repaint fast enough so that it does not appear to be slow?

#### **Behavior Problems**

Guidance about dealing with various problems in the application.

In addition to crashes, various behavior-related problems can occur. Some of these problems are listed below. Their descriptions can guide you to the Java SE Desktop technology to troubleshoot.

- Hangs occur when the application stops responding to user input. See Troubleshoot Process Hangs and Loops.
- **Exceptions in Java code** are visibly thrown to the console or the application log files. An examination of this output will guide you to the problem area.
- **Rendering and repainting issues** indicate a problem in Java 2D or in Swing. For example, the application's appearance is incorrect after a repaint that was caused by another application being dragged over it. Other examples are incorrect font, wrong colors, scrolling, damaging the application's frame by dragging another window over it, and updating a damaged area.

A quick test is the following: If the problem is reproducible on a different platform (for example, the problem was originally seen on Windows, and it is also present on Oracle Solaris or Linux), it is very likely to be a Swing PaintManager problem.

For the ways to change the Java 2D rendering pipelines with some flags, see Java 2D. This can also help determine if the problem is related to Java 2D or to Swing.

Multiscreen-related repainting issues belong to Java 2D (for example, repainting problems when moving a window from one screen to another, or other unusual behavior caused by the interaction with a non-default screen device).

- Issues related to desktop interaction indicate a problem in AWT. Some examples of such issues occur when moving, resizing, minimizing and maximizing windows, handling focus, enumerating multiple screens, using modality, interacting with the notification area (system tray), and viewing splash screens.
- Drag-and-drop problems are related to AWT.
- Printing problems could be related either to Java 2D or AWT depending on the API that is used.
- **Text-rendering issues** in AWT applications might be a problem in font properties or in internationalization.



However, if your application is purely AWT, text rendering problems might also be caused by Java 2D. On Oracle Solaris or Linux, text rendering is performed by Java 2D.

Text rendering in Swing is performed by Java 2D. Therefore, if your application uses Swing and you have text rendering problems (such as missing glyphs, incorrect rendering of glyphs, incorrect spacing between lines or characters, bad quality of font rendering), then the problem is likely to be in Java 2D.

- Painting problems are most likely a Swing issue.
- Full-screen issues are related to the Java 2D API.
- Encoding and locales issues (for example, no locale-specific characters displayed) indicate internalization problems.

# **Basic Tools**

List of basic tools that can help troubleshoot certain types of issues.

This section lists a few tools that can help you troubleshoot certain types of issues.

- **Performance:** Benchmarks, profilers, DTrace, Java probe.
- FootPrint: jmap, profilers
- Crashes: Native debuggers
- Hangs: JConsole, jstack, Control+Break
- Font-rendering: Font2DTest (delivered with the JDK 8 demos and samples bundle in the demo/jfc/Font2DTest directory)

## Java Debug Wire Protocol

The Java Debug Wire Protocol (JDWP) is very useful for debugging applications.

To debug an application using JDWP:

- 1. Open the command line, and set the PATH environment variable to *jdk*/bin where *jdk* is the installation directory of the JDK.
- Use the following command to run the application (called Test in this example) that you want to debug:
  - On Windows:

```
java -Xdebug -
Xrunjdwp:transport=dt_shmem,address=debug,server=y,suspend=y Test
```

On Oracle Solaris and Linux operating systems:

```
java -Xdebug -
Xrunjdwp:transport=dt socket,address=8888,server=y,suspend=y Test
```

The Test class will start in the debugging mode and wait for a debugger to attach to it at address debug (on Windows) or 8888 (on Oracle Solaris and Linux operating systems).



- 3. Open another command line, and use the following command to run jdb and attach it to the running debug server:
  - On Windows:

jdb -attach 'debug'

On Oracle Solaris and Linux operating systems:

jdb -attach 8888

After jdb initializes and attaches to Test, you can perform Java-level debugging.

4. Set your breakpoints and run the application. For example, to set the breakpoint at the beginning of the main method in Test, run the following command:

stop in Test.main run

When the jdb utility hits the breakpoint, you will be able to inspect the environment in which the application is running and see if it is functioning as expected.

- 5. (Optional) To perform native-level debugging along with Java-level debugging, use native debuggers to attach to the Java process running with JDWP.
  - On Oracle Solaris, you can use the dbx utility and on Linux, you can use the gdb utility.
  - On Windows, you can use Visual Studio for native-level debugging as follows:
    - a. Open Visual Studio.
    - **b.** On the **Debug** menu, select **Attach to Process**. Select the Java process that is running with JDWP.
    - c. On the **Project** menu, select **Settings**, and open the **Debug** tab. In the **Category** drop-down list, select **Additional DLLs** and add the native DLL that you want to debug (for example, Test.dll).
    - d. Open the source file (one or more) of Test.dll and set your breakpoints.
    - e. Enter cont in the jdb window. The process will hit the breakpoint in Visual Studio.



# 10 AWT

This chapter provides information and guidance about some specific procedures for troubleshooting common issues that might occur in the Java SE Abstract Window Toolkit (AWT).

This chapter contains the following sections:

- Debug Tips for AWT
- Layout Manager Issues
- Key Events
- Modality Issues
- AWT Crashes
- Focus Events
- Data Transfer
- Other Issues
- Heavyweight or Lightweight Components Mix

# Debug Tips for AWT

Helpful tips to debug issues related to AWT.

To dump the AWT component hierarchy, press Control+Shift+F1.

If the application hangs, get a stack trace by pressing Control+Break on Windows (which sends the SIGBREAK signal) or Control+\ on the Oracle Solaris and Linux operating systems (which sends the SIGQUIT signal).

To trace X11 errors on the Oracle Solaris and Linux operating systems, set the sun.awt.noisyerrorhandler system property to true. In Java SE 6 and earlier releases, the NOISY\_AWT environment variable was used for this purpose.

Before Java SE 8, exceptions thrown in the AWT Event Dispatch Thread (EDT) could be caught by setting the system property sun.awt.exception.handler to the name of the class that implements the public void handle(Throwable) method. This mechanism was updated in Java SE 8 to use the standard Thread.UncaughtExceptionHandler interface.

Loggers can produce helpful output when debugging AWT problems. See java.util.logging package description.

The following loggers are available:

java.awt java.awt.focus java.awt.event java.awt.mixing sun.awt



sun.awt.windows
sun.awt.X11

## Layout Manager Issues

This section describes possible problems with layout managers and provides workarounds when available.

The following problems occur with layout managers and workarounds:

#### 1. Call to invalidate () and validate() increases component size

**Cause:** Due to some specifics of the GridBagLayout layout manager, if ipadx or ipady is set, and invalidate() and validate() are called, then the size of the component increases to the value of ipadx or ipady. This happens because the GridBagLayout layout manager iteratively calculates the amount of space needed to store the component within the container.

**Workaround:** The JDK does not provide a reliable and simple way to detect if the layout manager should rearrange components or not in such a case, but there is a simple workaround. Use components with the overridden method getPreferredSize(), which returns the current size needed, as shown in the following example.

```
public Dimension getPreferredSize() {
    return new Dimension(size+xpad*2+1, size+ypad*2+1);
}
```

2. Infinite recursion with validate() from any Container.doLayout() method

**Cause:** Invoking validate() from any Container.doLayout() method can lead to infinite recursion because AWT itself invokes doLayout() from validate().

# **Key Events**

Issues related to handling key events that do not have a solution in the current release.

The following keyboard issues are currently unresolved:

- On some non-English keyboards, certain accented keys are engraved on the key and therefore are primary layer characters. Nevertheless, they cannot be used for mnemonics because there is no corresponding Java keycode.
- Changing the default locale at runtime does not change the text that is displayed for the menu accelerator keys.
- On a standard 109-key Japanese keyboard, the yen key and the backslash key both generate a backslash, because they have the same character code for the WM CHAR message. AWT should distinguish them.

The following keyboard issues concern the Oracle Solaris 10 and Linux x86 systems.

 Keyboard input in these systems is usually based on the X keyboard extension (XKB) of the X Window System. Users can configure one keyboard layout (for



instance, Danish: dk) or several layouts to switch between (for example, us and dk).

- With some keyboard layouts, for instance sk, hu, and cz, pressing the decimal separator on the numeric keypad not only enters a delimiter but also deletes the previous character. This is due to a native bug. A workaround is to use two layouts, for example, us and sk. In this case, the numeric keypad works correctly in both layouts.
- On UNIX systems that support dynamic keyboard changes, a running Java application does not recognize such a change. For instance, changing the keyboard from US to German does not change the keyboard mapping. Although the X server detects the change and sends out a MappingNotify event to interested clients AWT does not refresh its notion of the keycode-keysym mapping.

# **Modality Issues**

Information about issues related to using modality.

With the Java SE 6 release, many problems were fixed and many improvements were implemented in the area of AWT modality. If you see a modality problem with Java SE 1.5 or an earlier release, first upgrade to the latest Java SE release to see if the problem was already fixed.

Some of the problems that were fixed in Java SE 6 are the following:

- A modal dialog box goes behind a blocked frame.
- Two modal dialog boxes with the same parent window opened at the same time.

The section addresses the following issues.

#### • UNIX window managers:

Many of the modality improvements are unavailable in some Oracle Solaris or Linux environments, for example, when using Common Desktop Environment (CDE) window managers. With Java SE 6 and later releases, to see if a modality type or modal exclusion type is supported in a particular configuration, use the following methods:

- Toolkit.isModalityTypeSupported()
- Toolkit.isModalExclusionTypeSupported()

When a modal dialog box appears on the screen, the window manager might hide some of the Java top-level windows in the same application from the taskbar. This can confuse end users, but it does not affect their work much, because all the hidden windows are modal blocked and cannot be operated.

#### Other modality problems:

For more information about modality-related features and how to use them, see the AWT Modality specification.

One of the sections in that specification describes some AWT features that might be related to or affected by modal dialog boxes: always-on-top property, focus handling, window states, and so on. Application behavior in such cases is usually unspecified or depends on the platform; therefore, do not rely on any particular behavior.

## **AWT Crashes**

This section shows you how to identify and troubleshoot crashes related to AWT.



#### • Distinguish an AWT crash:

When a crash occurs, an error log is created with information and the state obtained at the time of the crash. See Fatal Error Log.

A line near the top of the file indicates the library where the error occurred. The following example shows part of the error log file in the case when the crash was related to the AWT library.

```
# Java VM: Java HotSpot(TM) Client VM (1.6.0-beta2-b76 mixed mode,
sharing)
# Problematic frame:
# C [awt.dll+0x123456]
...
```

However, the crash can happen somewhere deep in the system libraries, although still caused by AWT. In such cases, the indication awt.dll does not appear as a problematic frame, and you need to look further in the file, in the section Stack: Native frames: Java frames as shown in the following example.

```
Stack: [0x0aeb0000,0x0aef0000), sp=0x0aeefa44, free space=254k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code,
C=native code)
C 0x00abc751
C [USER32.dll+0x3a5f]
C [USER32.dll+0x3b2e]
C [USER32.dll+0x5874]
C [USER32.dll+0x58a4]
C [ntdll.dll+0x108f]
C [USER32.dll+0x5e7e]
C [awt.dll+0xec889]
C [awt.dll+0xf877d]
j sun.awt.windows.WToolkit.eventLoop()V+0
j sun.awt.windows.WToolkit.run()V+69
j java.lang.Thread.run()V+11
v ~StubRoutines::call stub
V [jvm.dll+0x83c86]
V [jvm.dll+0xd870f]
V [jvm.dll+0x83b48]
V [jvm.dll+0x838a5]
V [jvm.dll+0x9ebc8]
V [jvm.dll+0x108ba1]
V [jvm.dll+0x108b6f]
C [MSVCRT.dll+0x27fb8]
C [kernel32.dll+0x202ed]
Java frames: (J=compiled Java code, j=interpreted, Vv=VM code)
j sun.awt.windows.WToolkit.eventLoop()V+0
j sun.awt.windows.WToolkit.run()V+69
j java.lang.Thread.run()V+11
v ~StubRoutines::call stub
```



If the text awt.dll appears somewhere in the native frames, then the crash might be related to AWT.

### Troubleshoot an AWT crash:

One of the possible causes of crashes is that many AWT operations are asynchronous. For example, if you show a frame with a call to frame.setVisible(true), then you cannot be sure that it will be the active window after the return from this call.

Another example concerns native file dialogs. It takes some time for the operating system to initialize and show these dialogs, and if you dispose of them immediately after the call to setVisible(true), then a crash might occur. Therefore, if your application contains some AWT calls running simultaneously or immediately one after another, it is a good idea to insert some delays between them or add some synchronization.

# **Focus Events**

The following sections discuss the troubleshooting issues related to focus events:

- How to Trace Focus Events
- Native Focus System
- Focus Models Supported by X Window Managers
- Miscellaneous Problems with Focus

### How to Trace Focus Events

You can trace focus events by adding a focus listener to the toolkit, as shown in the following example.

```
Toolkit.getDefaultToolkit().addAWTEventListener(new AWTEventListener(
    public void eventDispatched(AWTEvent e) {
        System.err.println(e);
    }
), FocusEvent.FOCUS_EVENT_MASK | WindowEvent.WINDOW_FOCUS_EVENT_MASK |
    WindowEvent.WINDOW EVENT MASK);
```

The System.err stream is used here because it does not buffer the output.



### NOT\_SUPPORTED:

The correct order of focus events is the following:

- FOCUS\_LOST on component losing focus
- WINDOW LOST FOCUS on top-level losing focus
- WINDOW DEACTIVATED on top-level losing activation
- WINDOW ACTIVATED on top-level becoming active widow
- WINDOW GAINED FOCUS on top-level becoming focused window
- FOCUS GAINED on component gaining focus

When focus is transferred between components inside the focused window, only FOCUS\_LOST and FOCUS\_GAINED events should be generated. When focus is transferred between owned windows of the same owner or between an owned window and its owner, then the following events should be generated:

- FOCUS\_LOST
- WINDOW\_LOST\_FOCUS
- WINDOW\_GAINED\_FOCUS
- FOCUS\_GAINED

### Note:

The events losing focus or activation should come first.

### Native Focus System

Sometimes, a problem can be caused by the native platform. To check this, investigate the native events that are related to focus.

Ensure that the window you want to be focused gets activated and that the component you want to focus receives the native focus event.

On the Windows platform, the native focus events are the following:

- WM\_ACTIVATE for a top-level. WPARAM is WA\_ACTIVE when activating and WA INACTIVE when deactivating.
- WM SETFOCUS and WM KILLFOCUS for a component.

On the Windows platform, a concept of **synthetic focus** was implemented. It means that a focus owner component only emulates its focusable state, whereas real native focus is set to a **focus proxy** component. This component receives key and input method native messages and dispatches them to a focus owner. Before JDK7, a focus proxy component was a dedicated hidden child component inside a frame or dialog box. In the latest JDK releases a frame or dialog box serves as a focus proxy. Now, it proxies focus not only for components in an owned window but for all child components as well. A simple window never receives native focus and relies on the



focus proxy of its owner. This mechanism is transparent for a user but should be taken into account when debugging.

On Oracle Solaris and Linux operating systems, XToolkit uses a focus model that allows AWT to manage focus itself. With this model the window manager does not directly set input focus on a top-level window, but instead it sends only the WM\_TAKE\_FOCUS client message to indicate that focus should be set. AWT then explicitly sets focus on the top-level window if it is allowed.

### Note:

The X server and some window managers may send focus events to a window. However, these events are discarded by AWT.

AWT does not generate the hierarchical chains of focus events when a component inside a top-level gains focus. Moreover, the native window mapped to the component does not get a native focus event. On the Oracle Solaris and Linux platforms, as well as on the Windows platform, AWT uses the focus proxy mechanism. Therefore, focus on the component is set by synthesizing a focus event, whereas the invisible focus proxy has native focus.

A native window that is mapped to a Window object (not a Frame or Dialog object) has the override-redirect flag set. Thus, the window manager does not notify the window about the focus change. Focus is requested on the window only in response to a mouse click. This window will not receive native focus events at all. Therefore, you can trace only FocusIn or FocusOut events on a frame or dialog box. Because the major processing of focus occurs at the Java level, debugging focus with XToolkit is simpler than with WToolkit.

### Focus Models Supported by X Window Managers

The following focus models are supported by X window managers:

- Click-to-focus is a commonly used focus model. (For example, Microsoft Windows uses this model.)
- Focus-follows-mouse is a focus model in which focus goes to the window that the mouse hovers over.

### Miscellaneous Problems with Focus

This section discusses issues related to focus in AWT that can occur and suggested solutions.

1. Linux + KDE, XToolkit cannot be switched between two frames when a frame's title is clicked.

Clicking a component inside a frame causes the focus to change.

Solution: Check the version of your window manager and upgrade it to 3.0 or greater.

2. You want to manage focus using KeyListener to transfer the focus in response to Tab/Shift+Tab, but the key event doesn't appear.

**Solution:** To catch traversal key events, you must enable them by calling Component.setFocusTraversalKeysEnabled(true).



### A window is set to modal excluded with Window.setModalExclusionType (ModalExclusionType).

The frame, its owner, is modal blocked. In this case, the window will also remain modal blocked.

**Solution:** A window cannot become the focused window when its owner is not allowed to get focus. The solution is to exclude the owner from modality.

# 4. On Windows, a component requests focus and is concurrently removed from its container.

Sometimes java.lang.NullPointerException: null pData is thrown.

**Solution:** The easiest way to avoid throwing the exception is to do the removal along with requesting focus on EDT. Another, more complicated approach is to synchronize the requesting focus and removal if you need to perform these actions on different threads.

# 5. When focus is requested on a component and the focus owner is immediately removed, focus goes to the component after the removed component.

For example, Component A is the focus owner. Focus is requested on Component B, and immediately after this Component A is removed from its container. Eventually, focus goes to Component C, which is located after Component A in the container, but not to Component B.

**Solution:** In this case, ensure that the requesting focus is executed after Component A is removed, not before.

# 6. On Windows, when a window is set to alwaysOnTop in an inactive frame, the window cannot receive key events.

For example, a frame is displayed with a window that it owns. The frame is inactive, so the window is not focused. Then, the window is set to alwaysOnTop. The window gains focus, but its owner remains inactive. Therefore, the window cannot receive key events.

**Solution:** Bring the frame to the front (the Frame.toFront() method) before setting the window to alwaysOnTop.

# 7. When a splash screen is shown and a frame is shown after the splash screen window closes, the frame does not get activated.

**Solution:** Bring the frame to the front (the Frame.toFront() method) after showing it (the Frame.setVisible(true) method).

# 8. The WindowFocusListener.windowGainedFocus(WindowEvent) method does not return the frame's most-recent focus owner.

For example, a frame is the focused window, and one of its components is the focus owner. Another window is clicked, and then the frame is clicked again. WINDOW\_GAINED\_FOCUS comes to the frame and the WindowFocusListener.windowGainedFocus(WindowEvent) method is called. However, inside of this callback, you cannot determine the frame's mostrecent focus owner, because Frame.getMostRecentFocusOwner() returns null.

Solution: You can get the frame's most recent focus owner inside the WindowListener.windowActivated(WindowEvent) callback. However, by



this time, the frame will have become the focused window only if it does not have owned windows.

### Note:

This approach does not work for the window, only for the frame or dialog box.

9. A window is disabled with Component.setEnabled(false), but is not get completely unfocusable.

Solution: Do not assume that the condition set by calling Component.setEnabled(false) Or Component.setFocusable(false) will be maintained unfocusable along with all its content. Instead, use the Window.setFocusableWindowState(boolean) method.

# Data Transfer

The following sections discuss possible problems with data transfer features, which allow you to add *drag-and-drop* (DnD) and *cut*, *copy*, *and paste* (CCP) operations to the application.

- Debug Drag-and-Drop Applications
- Frequent Issues with Data Transfer

# Debug Drag-and-Drop Applications

It is difficult to use a debugger to troubleshoot DnD features, because during the drag-anddrop operation all input is grabbed. Therefore, if you place a breakpoint during DnD, you might need to restart your X server. Try to use remote debugging instead.

Two simple methods can be used to troubleshoot most issues with DnD:

- Printing all DataFlavor instances
- Printing received data

An alternative to remote debugging is the System.err.println() function, which prints output without delay.

### Frequent Issues with Data Transfer

This section describes issues that frequently occur with data transfer operations in AWT and suggests troubleshooting solutions.

1. Pasting a large amount of data from the clipboard takes too much time.

Using the Clipboard.getContents() function for a paste operation sometimes causes the application to hang for a while, especially if a rich application provides the data to paste.

The Clipboard.getContents() function fetches clipboard data in all available types (for example, some text and image types), and this can be expensive and unnecessary.



**Solution:** Use the Clipboard.getData() method to get only specific data from the clipboard. If data in only one or a few types are needed, then use one of the following Clipboard methods instead of getContents():

- DataFlavor[] getAvailableDataFlavors()
- boolean isDataFlavorAvailable(DataFlavor flavor)
- Object getData(DataFlavor flavor)
- 2. When a Java application uses Transferable.getTransferData() for DnD operations, the drag seems to take a long time.

In order to initialize transferred data only if it is needed, the initialization code was put in Transferable.getTransferData().

Transferable data is expensive to generate, and during a DnD operation Transferable.getTransferData() is invoked more than once, causing a slowdown.

Solution: Cache the Transferable data so that it is generated only once.

# 3. Files cannot be transferred between a Java application and the GNOME/KDE desktop and file browser.

On Windows and some window managers, transferred file lists can be represented as the DataFlavor.javaFileListFlavor data type. But, not all window managers represent lists of files in this format. For example, the GNOME window manager represents a file list as a list of URIS.

**Workaround:** To get files, request data of type String, and then translate the string to a list of files according to the text/uri-list format described in RFC 2483. To enable dropping files from a Java application to GNOME/KDE desktop and file browser, export data in the text/uri-list format. For an example, see the Work Around section from the RFE.

4. An image is passed to one of the startDrag() methods of DragGestureEvent Of DragSource, but the image is not displayed during the subsequent DnD operation.

**Solution:** Move a window with an image rendered on it as the mouse cursor moves during a DnD operation. See the code example in the Work Around section from the RFE.

#### 5. There is no way to transfer an array using DnD.

The DataFlavor class has no constructor that handles arrays. The mime type for an array contains characters that escapes. The code in the following example throws an IllegalArgumentException.

```
new DataFlavor(DataFlavor.javaJVMLocalObjectMimeType +
"; class=" +
  (new String[0]).getClass().getName())
```

**Solution:** "Quote" the value of the representation class parameter, as shown in the following example, where the quotation marks escape:

```
new DataFlavor(DataFlavor.javaJVMLocalObjectMimeType +
"; class=" +
"\"" +
```



```
(new String[0]).getClass().getName() +
"\"")
```

See bug report.

6. There are problems using AWT DnD support with Swing components.

Various problems can happen, for example, odd events are fired during a DnD operation, multiple items cannot be dragged and dropped, an InvalidDnDOperationException is thrown.

**Solution:** Use Swing's DnD support with Swing components. Although the Swing DnD implementation is based on the AWT DnD implementation, you cannot mix Swing and AWT DnD. See Lesson: Drag and Drop and Data Transfer in the Java Tutorials.

### 7. There is no way to change the state of the source to depend on the target.

In order to change the state of the source to depend on the target, you must have references to the source and target components in the same area of code, but this is not currently implemented in the DnD API.

**Workaround:** One workaround is to add flags to the transferable object that allow you to determine the context of the event.

For the transfer of data within one Java VM, the following workaround is proposed:

- Implement your target component as DragSourceListener.
- In DragGestureRecognizer.dragGestureRecognized(), add the target at the drag source listener, as shown in the following example.

• Now you can get the target and the source in the dragEnter(), dragOver(), dropActionChanged(), and dragDropEnd() methods of DragSourceListener().

### 8. Transferring objects in an application takes a long time.

The transferring of a big bundle of data or the creation of transferred objects takes too long. The user must wait a long time for the data transfer to complete.

This expensive operation makes transferring too long because you must wait until Transferable.getTransferData() finishes.

**Solution:** This solution is valid only for transferring data within one Java VM. Create or get expensive resources before the drag operation. For example, get the file content when you create a transferable data, so that Transferable.getTransferData() will not be too long.

# Other Issues

The following subsections discuss troubleshooting tips for other issues:



- Splash Screen Issues
- Tray Icon Issues
- Pop-up Menu Issues
- Background or Foreground Color Inheritance
- AWT Panel Size Restriction
- Hangs During Debugging of Pop-up Menus and Similar Components on X11
- Window.toFront()/toBack() Behavior on X11

## Splash Screen Issues

Issues that can happen with splash screen AWT and solutions.

This section describes some issues that can happen with the splash screen in AWT:

1. The user specified a JAR file with an appropriate MANIFEST.MF in -classpath, but the splash screen does not work.

**Solution:** See the solution for the next issue.

2. It is not clear which of several JAR files in an application should contain the splash screen image.

**Solution:** The splash screen image will be picked from a JAR file only if the file is used with the -jar command-line option. This JAR file should contain both the "SplashScreen-Image" manifest option and the image file. JAR files in -classpath will never be checked for splash screens in MANIFEST.MF. If you do not use -jar, you can still use -splash to specify the splash screen image in the command line.

3. Translucent PNG splash screens do not work on the Oracle Solaris and Linux operating systems.

**Solution:** This is a native limitation of X11. On the Oracle Solaris and Linux operating systems, the alpha channel of a translucent image will be compared with the 50% threshold. Alpha values above 0.5 will make opaque pixels, and pixels with alpha values below 0.5 will be completely transparent.

### Tray Icon Issues

If a SecurityManager is installed, then the value of AWTPermission must be set to accessSystemTray in order to create a TrayIcon object.

### Pop-up Menu Issues

In the JPopupMenu.setInvoker() method, the invoker is the component in which the pop-up menu is to be displayed. If this property is set to null, then the pop-up menu does not function correctly.

The solution is to set the pop-up menu's invoker to itself.



# Background or Foreground Color Inheritance

To ensure the consistency of your application on every platform, use explicit color assignment (both foreground and background) for every component or container.

Many AWT components use their own defaults for background and foreground colors instead of using parent colors.

This behavior is platform-dependent; the same component can behave differently on different platforms. In addition, some components use the default value for one of the background or foreground colors, but take the value from the parent for another color.

## AWT Panel Size Restriction

The AWT container has a size limitation. On most platforms, this limit is 32,767 pixels.

This means that, for example, if the canvas objects are 25 pixels high, then a Java AWT panel cannot display more than 1310 objects.

Unfortunately, there is no way to change this limit, neither with Java code nor with native code. The limit depends on what data type the operating system uses to store the widget size. For example, the Linux X windows system use the integer type, and are therefore limited to the maximum size of an integer. Other operating systems might use different types, such as long, and in this case, the limit could be higher.

See the documentation for your platform.

The following are examples of workarounds for this limit that might be helpful:

- Display components, page by page.
- Use tabs to display a few components at a time.

# Hangs During Debugging of Pop-up Menus and Similar Components on X11

Set the -Dsun.awt.disablegrab=true system property during the debugging of certain graphical user interface (GUI) components.

Certain graphical user interface (GUI) actions require grabbing all the input events in order to determine when the action should terminate (for example, navigating pop-up menus). While the grab is active, no other applications receive input events. If a Java application is being debugged, and a breakpoint is reached while the grab is active, then the operating system appears to hang. This happens because the Java application holding the grab is stopped by the debugger and cannot process any input events, and other applications do not receive the events due to the installed grab. In order to allow debugging such applications, the following system property should be set when running the application from the debugger:

-Dsun.awt.disablegrab=true

This property effectively turns off setting the grab, and does not hang the system. However, with this option set, in some cases, this can lead to the inability to terminate a GUI actions that would normally be terminated. For example, pop-up menus may not be dismissed when clicking a window's title bar.



# Window.toFront()/toBack() Behavior on X11

Due to restrictions enforced by third-party software (in particular, by window managers such as the Metacity), the toFront()/toBack() methods may not work as expected and cause the window to not change its stacking order in relation to other top-level windows.

More details are available in the CR 6472274.

If an application wants to bring a window to the top, it can try to workaround the issue by calling Window.setAlwaysOnTop(true) to temporarily make the window always stay on top and then calling setAlwaysOnTop(false) to reset the "always on top" state.

### Note:

This workaround is not guaranteed to work because window managers can enforce more restrictions. Also, setting a window to "always on top" is available to trusted applications only.

However, native applications experience similar issues, and this peculiarity makes Java applications behave similar to native applications.

# Heavyweight or Lightweight Components Mix

The following issues are addressed in the heavyweight or lightweight (HW/LW) component mixing feature:

### • Validate the component hierarchy:

Changing any layout-related properties of a component, such as its size, location, or font, invalidates the component as well as its ancestors. In order for the HW/LW Mixing feature to function correctly, the component hierarchy must be validated after making such changes. By default, invalidation stops on the top-most container of the hierarchy (for example, a Frame object). Therefore, to restore the validity of the hierarchy, the application should call the Frame.validate() method. For example:

```
component.setFont(myFont);
frame.validate();
```

frame refers to a frame that contains component.



### Note:

Swing applications and the Swing library often use the following pattern:

```
component.setFont(myFont);
component.revalidate();
```

The revalidate() call is *not* sufficient because it validates the hierarchy starting from the nearest validate root of the component only, thus leaving the upper containers invalid. In that case, the HW/LW feature may not calculate correct shapes for the HW components, and visual artifacts may be seen on the screen.

To verify the validity of the whole component hierarchy, a user can use the key combination Control+Shift+F1, as described in Debug Tips for AWT. A component marked invalid may indicate a missing validate() call somewhere.

#### • Validate roots:

The concept of validate roots mentioned in **Validate the component hierarchy** was introduced in Swing in order to speed up the process of validating component hierarchies because it may take a significant amount of time. While such optimization leaves upper parts of hierarchies invalid, this did not create any issues because the layout of components inside a validate root does not affect the layout of the outside component hierarchy (that is, the siblings of the validate root). However, when HW and LW components are mixed together in a hierarchy, this statement is no longer true. That is why the feature requires the whole component hierarchy to be valid.

Calling frame.validate() may be inefficient, and AWT supports an alternative, optimized way of handling invalidation/validation of component hierarchies. This feature is enabled with a system property:

-Djava.awt.smartInvalidate=true

Once this property is specified, the invalidate() method will stop invalidation of the hierarchy when it reaches the nearest validate root of a component on which the invalidate() method has been invoked. Afterward, to restore the validity of the component hierarchy, the application should simply call:

component.revalidate();

### Note:

In this case, calling frame.validate() would be effectively a no-op (a statement that does nothing) because frame is still valid. Since some applications rely on calling validate() directly on a component upper than the validate root of the hierarchy (for example, a frame), this new optimized behavior may cause incompatibility issues, and hence it is available only when specifying the system property.



If an application experiences any difficulties running in this new optimized mode, a user can use the key combination Control+Shift+F1 as described in Debug Tips for AWT to investigate what parts of the component hierarchy are left invalid, and thus possibly cause the problems.

### • Swing painting optimization:

By default, the Swing library assumes that there are no HW components in the component hierarchy, and therefore uses optimized drawing techniques to boost performance of the Swing GUI. If a component hierarchy contains HW components, the optimizations must be turned off. This is relevant for Swing JScrollPanes in the first place. You can change the scrolling mode by using the JViewPort.setScrollMode(int) method.

### • Non-opaque LW components:

Non-opaque LW components are not supported by the HW/LW mixing feature implementation by default. In order to enable mixing non-rectangular LW components with HW components, the application must use the com.sun.awt.AWTUtilities.setComponentMixingCutoutShape() non-public API.

### Note:

The non-rectangular LW components should still paint themselves using either opaque (alpha = 1.0) or transparent (alpha = 0.0) colors. Using translucent colors (with 0.0 <alpha < 1.0) is not supported.

### • Disable the default HW/LW mix feature:

In the past, some developers have implemented their own support for cases when HW and LW components must be mixed together. In order to disable the built-in feature the application must be started with the following system property:

-Dsun.awt.disableMixing=true



# 11

# Java 2D Pipeline Rendering and Properties

This chapter provides information and guidance for troubleshooting some of the most common issues that might be found in the Java 2D API when changing pipeline rendering and properties.

For a summary of Java 2D properties, see Java 2D Properties.

By choosing a different pipeline, or manipulating the properties of a pipeline, you might be able to determine the cause of the problem, and often find a workaround.

In general, you can trouble shoot Java 2D pipeline issues by determining the default pipeline used in your configuration. Then, either change the pipeline to another one, or modify the properties of the default pipeline.

If the problem disappears, then you found a workaround. If the problem persists, then try changing another property or pipeline.

Java 2D uses a set of pipelines, which can be roughly defined as different ways of rendering the primitives. These pipelines are as follows:

- Oracle Solaris and Linux: X11 Pipelineis the default for the Oracle Solaris and Linux operating systems.
- Windows OS: DirectDraw/GDI Pipeline is the default on Windows
- Windows OS: Direct3D Pipeline in Full-Screen Mode is an alternative on Windows.
- OpenGL Pipeline in Oracle Solaris, Linux, and Windowsis an alternative on the Oracle Solaris and Linux operating systems, as well as Windows.

# Oracle Solaris and Linux: X11 Pipeline

On UNIX platforms, the default pipeline is the X11 pipeline. This pipeline uses the X protocol for rendering to the screen or to certain types of offscreen images, such as VolatileImages, or "compatible" images (images that are created with the GraphicsConfiguration.createCompatibleImage() method).

These types of images can be put into X11 pixmaps for improved performance, especially in the case of the Remote X server.

In addition, in certain cases, Java 2D uses X server extensions, for example, the MIT X shared memory extension, or Direct Graphics Access extension, Double-buffer extension for double-buffering when using the BufferStrategy API.

An additional pipeline, the OpenGL pipeline, might offer greater performance in some configurations.

The following are X11 pipeline properties to troubleshoot.

- X11 Pipeline Pixmaps Properties
- X11 Pipeline MIT Shared Memory Extension
- Oracle Solaris on SPARC: DGA Support



• Oracle Solaris on SPARC - Change Java 2D Default Visual

# X11 Pipeline Pixmaps Properties

Java 2D by default uses X11 pixmaps for storing or caching certain types of offscreen images.

Only the following types of images can be stored in pixmaps:

- Opaque images, in which case ColorModel.getTransparency() returns Transparency.OPAQUE
- 1-bit transparent images (also known as sprites, Transparency.BITMASK)

The advantage of using pixmaps for storing images is that they can be put into the framebuffer's video memory at the driver's discretion, which improves the speed at which these pixmaps can be copied to the screen or another pixmap.

The use of pixmaps typically results in better performance. However, in certain cases, the opposite is true. These cases typically involve the use of operations that cannot be performed using the X protocol, such as antialiasing, alpha compositing, and transforms that are more complex than simple translation transforms.

For these operations, the X11 pipeline must do the rendering using the built-in software renderer. In most cases, this includes reading the contents of the pixmap to system memory (over the network in the case of remote X server), performing the rendering, and then sending the pixels back to the pixmap. These operations could result in extremely poor performance, especially if the X server is remote.

The following are two cases to disable the use of X11 pipeline:

#### Disable X11 pipeline pixmaps:

To disable the use of pixmaps by Java2D, pass the following property to the Java VM: -Dsun.java2d.pmoffscreen=false.

### • Disable X11 pipeline shared memory pixmaps:

To minimize the effect of operations that require reading pixels from a pixmap on overall performance, the X11 pipeline uses shared memory pixmaps for storing images that are often read from.

### Note:

The shared memory pixmaps can only be used in the case of a local X server.

The advantage of using shared memory pixmaps is that the pipeline can get direct access to the pixels in the pipeline, bypassing the X11 protocol, which results in better performance.

By default, an image is stored in a normal X server pixmap, but it can be later moved to a shared memory pixmap if the pipeline detects excessive reading from such an image. The image can be moved back to a server pixmap if it is copied from often enough.

The pipeline allows two ways of controlling the use of shared memory pixmaps: either disabling them or forcing all images to be stored in shared memory pixmaps.



First, try forcing the shared memory pixmaps because it often improves performance. However, with certain video board/driver configurations, it may be necessary to disable the shared memory pixmaps to avoid rendering artifacts or crashes.

- To disable shared memory pixmaps, set the J2D\_PIXMAPS environment variable to server. This is the default in remote X server case.
- To force all pixmaps to be created in shared memory, set J2D PIXMAPS to shared.

## X11 Pipeline MIT Shared Memory Extension

The Java 2D X11 pipeline uses the MIT Shared Memory Extension (MIT SHM), which allows a faster exchange of data between the client and the X server. This can significantly improve the performance of Java applications.

The following are two ways to improve the performance of the Java application.

• Increase X Server and Java 2D shared memory:

On the Oracle Solaris operating system releases 8 and earlier, it was sometimes necessary to increase the amount of shared memory available to the system (and to X server in particular) because the default was too low, resulting in poor rendering performance. Increasing the amount of shared memory and shared memory segments can result in better performance.

To change the default settings on the Oracle Solaris operating system, edit the /etc/ system file and change the shmsys:shminfo\_\* settings, as shown in the following example. Note that this is not needed on Oracle Solaris 9 and later.

```
set shmsys:shminfo_shmmax=10000000
set shmsys:shminfo_shmni=200
set shmsys:shminfo_shminfo=150
```

On Linux, this setting can be configured by editing the /proc/sys/kernel/shm\* files.

• Disable X11 pipeline shared memory extension:

In case of problems (such as crashes, or rendering artifacts) with older X servers and the Shared Memory Extension, it is useful to be able to disable the extension. To disable the use of MIT SHM, set the J2D USE MITSHM environment variable to false.

# Oracle Solaris on SPARC: DGA Support

On SPARC hardware, if the framebuffer supports Sun's Direct Graphics Access (DGA) X server extension, and Java 2D has a corresponding module for accessing the framebuffer, then DGA will be used for rendering to the screen.

All offscreen images will reside in Java heap memory, and Java 2D's software-only rendering pipeline is used for rendering to them. This is different from a typical UNIX configuration, where X11 pixmaps are used for offscreen images.

The following are use cases that describe how to detect DGA extension support and disable or enable DGA:

### DGA extension for rending

To detect if the DGA extension is used for rendering to the screen, run any Java application that does some rendering or displays a GUI, and check if a /tmp/wg\* file

was created when the application started. Exit the application and verify that the file was deleted. If this is the case, then on this system, Java 2D is using DGA.

#### • Typical DGA Issues:

Because DGA allows direct access to the framebuffer's video memory, the typical problems include corruption outside of window bounds, complete system, and X server lock-ups.

#### • Enable or Disable DGA:

If you determine that DGA is being used, the first thing to try is to disable it. This can be done by setting the NO\_J2D\_DGA environment variable to true. This forces the default UNIX path to use only X11 for rendering to the screen, and pixmaps for accelerating offscreen images.

Sometimes, it could be beneficial to enable the use of pixmaps, while also using DGA for rendering to the screen. To force the use of pixmaps for accelerating offscreen images, set the following property when starting the application: – Dsun.java2d.pmoffscreen=true.

## Oracle Solaris on SPARC - Change Java 2D Default Visual

On certain video boards on the SPARC platform, more than one visual can be available from the X server.

By default, Java 2D tries to select the best visual, where "best" is typically a higher-bit depth visual. For example, on some Oracle Solaris operating system releases, the default X11 visual is 8-bit PseudoColor, although 24-bit visual is also available. In these cases, Java 2D selects a 24-bit TrueColor visual as the default for Java windows.

While it is possible to create a Java top-level window with a GraphicsConfiguration object corresponding to a different visual, in some cases, it is necessary to make Java use a different default visual instead. This can be done by setting the FORCEDEFVIS environment variable. It can be set to true to force the use of the default X server visual (even if it is not the best one), or it can be set to a hexadecimal number corresponding to the visual ID as reported by tools like xdpyinfo.

To determine your X server default visual, execute the  ${\tt xdpyinfo}$  command and look at the default visual id field.

# Windows OS: DirectDraw/GDI Pipeline

The default pipeline on the Windows platform is a mixture of the DirectDraw pipeline and the GDI pipeline, where some operations are performed with the DirectDraw pipeline and others with the GDI pipeline. DirectDraw and GDI APIs are used for rendering to accelerated offscreen and onscreen surfaces.

The possible issues with the Direct3D pipeline include rendering artifacts, crashes, and performance related problems.

An additional pipeline, the OpenGL pipeline, might offer greater performance in some configurations.

The following are three cases to troubleshoot issues with the Direct3D pipeline such as rendering artifacts, crashes, and performance related problems:



### • Disable the DirectDraw pipeline:

When DirectDraw is disabled, all operations are performed with GDI. Provide the following flag to disable the use of DirectDraw: -Dsun.java2d.noddraw=true. In this case, all offscreen images will be created in the Java heap, and rendered with the default software pipeline. All onscreen rendering, as well as copies of offscreen images to the screen, will be performed using GDI.

#### Enable the DirectDraw pipeline:

If the pipeline was disabled by default for some reason, then it can be enabled by providing the -Dsun.java2d.noddraw=false flag to the VM.

However, typically there was a reason why it was disabled in the first place, so it is better not to force it.

#### Disable the built-in punting mechanism:

In general, the DirectDraw pipeline attempts to place the offscreen surfaces in the framebuffer's video memory, which provides the fast copies from these surfaces to the screen or other accelerated surfaces, as well as hardware accelerated rendering of certain graphics operations.

To limit the effect of unaccelerated rendering to VRAM-based surfaces, there exists a punting mechanism, which moves the surface that is detected to be often read from to the system memory. If the surface is found to be copied from often enough, it may be promoted back to video memory.

However, if the pipeline cannot perform an operation using the DirectDraw API (operations using, for example, alpha compositing, or transforms, or antialiasing), then rendering is performed using the software pipeline. In some cases, this means that the pixels of the destination surface, which resides in VRAM, must be read into system memory, which is a very expensive operation.

On certain video boards/drivers combinations, the system-memory-based DirectDraw surfaces are known to cause rendering artifacts and other issues. The DirectDraw pipeline provides a way to disable the punting mechanism so that the system memory surfaces are not used.

To defeat the built-in surface punting mechanism, provide the following flag to the Java VM: -Dsun.java2d.ddforcevram=true.

### Note:

This mechanism can result in performance degradation because the software loops may be reading pixels from VRAM on each operation. In this case, consider disabling the DirectDraw pipeline.

#### • Disable the DirectDraw BILT operations:

In a Bit Block Transfer (BILT) operation, two bitmap patterns are combined. This operation corresponds to a call to the Graphics.drawImage() API.

In some cases, it is possible to avoid rendering problems by disabling the DirectDraw BLIT operations. GDI BLITs will be used instead.



### Note:

This operation might result in bad performance. Consider disabling the DirectDraw pipeline instead.

To disable the use of DirectDraw BLIT operations, pass the parameter – Dsun.java2d.ddblit=false to the Java VM.

# Windows OS: Direct3D Pipeline in Full-Screen Mode

This pipeline is enabled in full-screen mode by default, if the drivers support the required features and the level of rendering quality.

It is possible to enable the Direct3D pipeline or to force its use, as described in the following sections.

Consider enabling the Direct3D pipeline for your application if it heavily uses rendering operations such as alpha compositing, antialiasing, and transforms.

However, use caution when deciding to enable this pipeline in your application. For example, some built-in video chipsets (which are used in most notebooks) do not perform well using Direct3D, even if they satisfy the quality requirements for Java 2D pipelines.

The following are three cases to troubleshoot problems with Direct3D API.

#### 1. Disable the Direct3D pipeline:

Some older video boards/drivers combinations are known to cause issues (both rendering and performance) with the Direct3D pipeline. To disable the pipeline in these cases, pass the parameter -Dsun.java2d.d3d=false to the Java VM, or set the J2D D3D environment variable to false.

### 2. Enable the Direct3D pipeline:

To enable the Direct3D pipeline in both windowed and full-screen mode, use the parameter -Dsun.java2d.d3d=true, or set the J2D\_D3D environment variable to true.

### Note:

The pipeline is enabled only if the drivers support the minimum required features.

#### 3. Diagnose the Direct3D pipeline rendering problems:

Some rendering issues (like missing pixels, garbled rendering) can be diagnosed by forcing different Direct3D rasterizers. Set the J2D\_D3D\_RASTERIZER environment variable to one of the following: ref, rgb, hal, or tnl.

See the Direct3D documentation for a description of these rasterizers. By default, the best rasterizer is chosen based on its advertised capabilities. In particular, the ref rasterizer forces the use of the reference Direct3D rasterizer from Microsoft. If a rendering problem is not reproducible with this rasterizer, then it is likely to be a video driver bug.



The rgb rasterizer is available only if the Direct3D SDK is installed. The Software Rasterizer for the Microsoft DirectX 9.0 Software Development Kit (SDK) is compatible with Microsoft Direct 3D. This can be obtained from Software Rasterizer for the Microsoft DirectX 9.0 Software Development Kit (SDK). Alternatively, download the Microsoft DirectX® End-User Runtime. For more information about the Direct3D SDK, see Enabling Support for the Direct3D Version 11 DDI.

For performance or quality problems with text rendering with the Direct3D pipeline, you can force the use of the ARGB texture instead of the default Alpha texture for the Direct3D pipeline's glyph cache. To do this, set the J2D\_D3D\_NOALPHATEXTURE environment variable to true.

# **OpenGL** Pipeline in Oracle Solaris, Linux, and Windows

The OpenGL pipeline is available on Oracle Solaris, Linux, and Windows.

This alternate pipeline uses the hardware-accelerated, cross-platform OpenGL API when rendering to VolatileImages, to backbuffers created with BufferStrategy API, and to the screen.

This pipeline can offer great performance advantages over the default (X11 or GDI/ DirectDraw) pipelines for certain applications. Consider enabling the pipeline for your application if it heavily uses of rendering operations like alpha compositing, antialiasing, and transforms.

The following are use cases for troubleshooting problems in OpenGL pipeline

- Enable OpenGL Pipeline
- Minimum Requirements
- Diagnose Startup Issues
- Diagnose Rendering and Performance Issues
- Latest OpenGL Drivers

### Enable OpenGL Pipeline

The OpenGL pipeline is disabled by default.

To attempt to enable the OpenGL pipeline, provide the following option to the JVM:

-Dsun.java2d.opengl=True

To receive verbose console output about whether the OpenGL pipeline is initialized successfully for a particular screen, set the option to True (note the uppercase *T*).

### **Minimum Requirements**

The OpenGL pipeline will not be enabled if the hardware or drivers do not meet the minimum requirements.

If one of the following requirements is not met, Java 2D will fall back and use the default pipeline (X11 on Oracle Solaris/Linux or GDI/DirectDraw on Windows), which means your application will continue to work correctly, but without the OpenGL acceleration.

The minimum requirements for the Oracle Solaris and Linux operating systems are the following:



- Hardware accelerated OpenGL/GLX libraries installed and configured properly
- OpenGL version 1.2 or higher
- GLX version 1.3 or higher
- At least one TrueColor visual with an available depth buffer

The minimum requirements for Windows OS are the following:

- Hardware accelerated drivers supporting the extensions WGL\_ARB\_pbuffer, WGL\_ARB\_render\_texture, and WGL\_ARB\_pixel\_format
- OpenGL version 1.2 or higher
- At least one pixel format with an available depth buffer

### Diagnose Startup Issues

You can get detailed information about the startup procedures of the OpenGL-based Java 2D pipeline by using the <code>J2D\_TRACE\_LEVEL</code> environment variable.

As previously mentioned, the OpenGL pipeline might not be enabled on certain machines for various reasons. For example, the drivers might not be properly installed and might report an insufficient version number. Alternatively, your machine might have an older graphics card that does not support the appropriate OpenGL version or extensions.

In the Java SE 6 and later releases, you can get detailed information about the startup procedures of the OpenGL-based Java 2D pipeline by using the J2D\_TRACE\_LEVEL environment variable, as shown in the following examples.

Set the J2D TRACE LEVEL environment variable on Windows.

```
# set J2D_TRACE_LEVEL=4
# java -Dsun.java2d.opengl=True YourApp
```

Set the J2D TRACE LEVEL environment variable on Solaris and Linux.

```
# export J2D_TRACE_LEVEL=4
# java -Dsun.java2d.opengl=True YourApp
```

The output will be different depending on your platform and the installed graphics hardware, but it can give you some insight into the reasons why the OpenGL pipeline is not being successfully enabled for your configuration.

### Note:

This output is especially useful when filing bug reports intended for the Java 2D team at Sun.



# **Diagnose Rendering and Performance Issues**

Diagnose if rendering or performance issues are being caused by Java 2D or by the OpenGL drivers.

Because the OpenGL pipeline relies so heavily on the underlying graphics hardware and drivers, it might sometimes be difficult to determine whether rendering or performance issues are being caused by Java 2D or by the OpenGL drivers.

One feature new to the OpenGL pipeline in the Java SE 6 release is the use of the GL\_EXT\_framebuffer\_object extension, which provides better performance for rendering and reduced VRAM consumption when using VolatileImages. This "FBO" codepath is enabled by default when the OpenGL pipeline is enabled, but only if your graphics hardware and driver support this OpenGL extension. This extension is generally available on Nvidia GeForce/Quadro FX series and later, and on ATI Radeon 9500 and later. If you suspect that the "FBO" codepath is causing problems in your application, then you can disable it by setting the following system property:

```
-Dsun.java2d.opengl.fbobject=false
```

Setting this property will cause Java 2D to fall back on the older pbuffer-based codepath.

If you find that a certain Java 2D operation causes different visual results with the OpenGL pipeline enabled than without, then it probably indicates a graphics driver bug. Similarly, if the performance of Java 2D rendering is significantly worse with the OpenGL pipeline enabled than without, then it is most likely caused by a driver or hardware problem.

In either case, file a detailed bug report through the normal bug reporting channels. See Submit a Bug Report. When filing bug reports, be as detailed as possible, and include the following information:

- Operating system (for example, Ubuntu Linux 6.06, Windows XP SP2)
- Name of graphics hardware manufacturer and device (for example, Nvidia GeForce 2 MX 440)
- Exact driver version (for example, ATI Catalyst 6.8, Nvidia 91.33)
- Output when J2D\_TRACE\_LEVEL=4 is specified on the command line (as described in the previous section)
- The output of the glxinfo command if you are on Oracle Solaris or Linux

# Latest OpenGL Drivers

Because the OpenGL pipeline relies heavily on the OpenGL API and the underlying graphics hardware and drivers, it is very important to ensure that you have the latest graphics drivers installed on your computer. The following table lists graphics card manufacturers with their corresponding supported platforms and some examples of cards that are known to support OpenGL.

Manufacturer	Platforms	Cards Known to Work
AMD	Linux, Windows	Radeon 8500 and later, FireGL series
Nvidia	Oracle Solaris on x64, Linux, Windows	GeForce 2 series and later, Quadro FX series and later



Manufacturer	Platforms	Cards Known to Work
Oracle	Oracle Solaris on SPARC	Expert3D series, XVR-500, XVR-600, XVR-1200, XVR-2500
Xi Graphics	Oracle Solaris on x86, Linux	Various (check with Xi Graphics)

# 12 Java 2D

This chapter provides information and guidance for troubleshooting some of the most common issues that might be found in the Java 2D API.

This chapter contains the following sections:

- Generic Performance Issues
- Text-Related Issues
- Java 2D Printing

For a summary of Java 2D properties, see Java 2D Properties.

# **Generic Performance Issues**

There could be many causes for poor rendering performance. The following topics identify the cause for your applications poor rendering performance and suggests some approaches to improve performance of software-only rendering.

This topic contains the following subsections:

- Hardware-Accelerated Rendering Primitives
- Primitive Tracing to Detect and Avoid Non-Accelerated Rendering
- Causes of Poor Rendering Performance
- Improve Performance of Software-only Rendering

### Hardware-Accelerated Rendering Primitives

In order to better understand what could be causing performance problems, take a look at what hardware acceleration means.

In general, hardware-accelerated rendering could be divided into two categories.

- Hardware-accelerated rendering to an "accelerated" destination. Examples of rendering destinations that can be hardware-accelerated are VolatileImage, screen and BufferStrategy. If a destination is accelerated, then rendering goes to a surface may be performed by video hardware. So, if you issue a drawRect call, Java 2D redirects this call to the underlying native API (such as GDI, DirectDraw, Direct3D or OpenGL, or X11), which performs the operation using hardware.
- Caching images in accelerated memory (video memory or pixmaps) so that they can be copied very fast to another accelerated surface. These images are known as managed images.

Ideally, all operations performed on an accelerated surface are hardware-accelerated. In this case, the application takes full advantage of what is offered by the platform.

Unfortunately in many cases the default pipelines are not able to use the hardware for rendering. This can happen due to the pipeline limitations, or the underlying native API. For



example, most X servers do not support rendering antialiased primitives, or alpha compositing.

One cause of performance issues is when operations performed are not hardwareaccelerated. Even in cases when a destination surface is accelerated, some primitives may not be.

It is important to know how to detect the cases when hardware acceleration is not being used. Knowing this may help in improving performance.

### Primitive Tracing to Detect and Avoid Non-Accelerated Rendering

To detect a non-accelerated rendering, you can use Java 2D primitive tracing.

Run your application with -Dsun.java2d.trace=count. When the application exits, a list of primitives and their counts is printed to the console.

Any time you see a MaskBlit or any of the General\* primitives, it typically means that some of your rendering is going through software loops. Here is the output from performing drawImage on a translucent BufferedImage to a VolatileImage on Linux:

sun.java2d.loops.Blit\$GeneralMaskBlit::Blit(IntArgb, SrcOverNoEa, "Integer BGR Pixmap")sun.java2d.loops.MaskBlit::MaskBlit(IntArgb, SrcOver, IntBgr)

Here are some of the common non-accelerated primitives in the default pipelines, and their signatures in the tracing output.

### Note:

Most of this tracing was taken on Linux; you may see some differences depending on your platform and configuration.

• Translucent images (images with ColorModel.getTranslucency() returnTranslucency.TRANSLUCENT), or images with AlphaCompositing. Sample primitive tracing output:

sun.java2d.loops.Blit\$GeneralMaskBlit::Blit(IntArgb,SrcOverNoEa,
"Integer BGR Pixmap")sun.java2d.loops.MaskBlit::MaskBlit(IntArgb,
SrcOver, IntBgr)

 Use of antialiasing (by setting the antialiasing hint). Sample primitive tracing output:

sun.java2d.loops.MaskFill::MaskFill(AnyColor, Src, IntBgr)

- Rendering antialiased text (setting the text antialising hint). Sample output can be one of the following:
  - sun.java2d.loops.DrawGlyphListAA::DrawGlyphListAA(OpaqueColor, SrcNoEa, AnyInt)



- sun.java2d.loops.DrawGlyphListLCD::DrawGlyphListLCD(AnyColor, SrcNoEa, IntBgr)
- Alpha compositing, either by rendering with translucent color (a color with an alpha value that is not 0xff) or by setting a non-default AlphaCompositing mode with Graphics2D.setComposite():

sun.java2d.loops.Blit\$GeneralMaskBlit::Blit(IntArgb, SrcOver, IntRgb)sun.java2d.loops.MaskBlit::MaskBlit(IntArgb, SrcOver, IntRgb)
]

• Non-trivial transforms (if the transform is more than only translation). Rendering a transformed opaque image to a VolatileImage:

sun.java2d.loops.TransformHelper::TransformHelper(IntBgr, SrcNoEa, IntArgbPre)

Rendering a rotated line:

sun.java2d.loops.DrawPath::DrawPath(AnyColor, SrcNoEa, AnyInt)

Run your application with tracing and ensure that you do not use unaccelerated primitives unless they are needed.

### Causes of Poor Rendering Performance

List of causes of poor rendering performance and possible alternatives.

Some of the possible causes of poor rendering performance and possible alternatives are described as follows:

Mixing accelerated and non-accelerated rendering:

A situation when only part of the primitives rendered by an application could be accelerated by the particular pipeline when rendering to an accelerated surface can cause thrashing, because the pipelines will be constantly trying to adjust for better rendering performance but with possibly little success.

If it is known beforehand that most of the rendering primitives will not be accelerated, then it could be better to either render to a BufferedImage and then copy it to the back buffer or the screen, or switch to a non-hardware accelerated pipeline using one of the flags discussed.

### Note:

This approach may limit your application's ability to take advantage of future improvements in Java 2D's use of hardware acceleration.

For example, if your application is often used in remote X server cases, but it heavily uses antialiasing, alpha compositing, and so forth, then the performance can be severely degraded. To avoid this, disable the use of pixmaps by setting the –



Dsun.java2d.pmoffscreen=false property either by passing it to the Java runtime, or by setting it programmatically using the System.setProperty() API.

### Note:

This property must be set before any GUI-related operations because it is read only once.

### Non-optimal rendering primitives:

It is preferable to use the simplest primitive possible to achieve the desired visual effect.

For example, use Graphics.drawLine() instead of new Line2D().draw(). The result looks the same. However, the second operation is much more computationally intensive because it is rendered as a generic shape, which is typically much more expensive to render. Shapes show up in different ways in the primitive tracing, depending on antialiasing settings and the specific pipeline, but most likely they will show up as many \*FillSpans or DrawPath primitives.

Another example of complicated attributes is GradientPaint. Although it may be hardware accelerated by some of the non-default pipelines (such as OpenGL), it is not hardware accelerated by the default pipelines. Therefore, you can restrict the use of GradientPaint if it causes performance problems.

### • Heap-based destination surface BufferedImage:

Rendering to a BufferedImage almost always uses software loops.

An exception on some SPARC systems is that the VIS instruction set can be used for accelerating certain imaging operations. See VIS Instruction Set.

To ensure that the rendering has the opportunity of being hardware accelerated, choose a BufferStrategy or a VolatileImage object as the rendering destination.

### Defeat built-in acceleration mechanism:

Java 2D attempts to accelerate certain types of images. The contents of images can be cached in video memory for faster copying to accelerated destinations such as <code>VolatileImages</code>. These mechanisms can be unknowingly defeated by the application.

• Get direct access to pixels with getDataBuffer():

If an application gets access to BufferedImage pixels by using the getRaster().getDataBuffer() API, then Java 2D will not be able to guarantee that the data in the cache is up to date, so it will disable any acceleration attempts of this type of image.

To avoid this, do not call getDataBuffer(). Instead, work with WriteableRaster, which can be obtained with the BufferedImage.getRaster() method.

If you need to modify the pixels directly, then you can manually cache your image in video memory by maintaining the cached copy of your image in a <code>VolatileImage</code>, and updating the cached data when the original image is touched.

Render to a sprite before every copy:



If an application renders to an image before copying it to an accelerated surface (VolatileImage, BufferStrategy), then the image cannot take advantage of being cached in accelerated memory. This is because the cached copy must be updated every time the original image is updated, and therefore only the default system-memory-based surface is used, and this means no acceleration.

### Exhausted accelerated memory resources:

If the application uses many images, then it can exhaust the available accelerated memory. If this is the cause of performance issues for your application, then you might need to handle the resources.

The following API can be used to request the amount of available accelerated memory: GraphicsDevice.getAvailableAcceleratedMemory().

In addition, the following API can be used to determine if your image is being accelerated: Image.getCapabilities().

If you determined that your application is exhausting the resources, you can handle the problem by not holding images you no longer need. For example, if your game advanced to the next level, release all images from the previous levels. You can also release accelerated resources associated with an image by using the <code>Image.flush()</code> API.

You can also use the acceleration priority API Image.getAccelerationPriority() and setAccelerationPriority() to specify the acceleration priority for your images. It is a good idea to make sure that at least your back-buffer is accelerated, so create it first, and with acceleration priority of 1 (default). You can also prohibit certain images from being accelerated if needed by setting the acceleration priority to 0.0.

### Improve Performance of Software-only Rendering

If your application relies on software-only rendering (by only rendering to a BufferedImage, or changing the default pipeline to an unaccelerated one), or even if it does mixed rendering, then the following are certain approaches to improving performance:

### 1. Image types or operations with optimized support:

Due to overall platform size constraints, Java 2D has a limited number of optimized routines for converting from one image format to another. In situations where an optimized direct loop can not be found, Java 2D will do the conversion through an intermediate image format (IntArgb). This results in performance degradation.

Java 2D primitive tracing can be used for detecting such situations.

For each drawImage call there will be two primitives: the first one converting the image from the source format to an intermediate IntArgb format and the second one converting from intermediate IntArgb to the destination format.

Here are two ways to avoid such situations:

- Use a different image format if possible.
- Convert your image to an intermediate image of one of the better-supported formats, such as INT\_RGB or INT\_ARGB. In this way the conversion from the custom image format will happen only once instead of on every copy.

### 2. Transparency vs translucency:

Consider using 1-bit transparent (BITMASK) images for your sprites as opposed to images with full translucency (such as INT\_ARGB) if possible.



Processing images with full alpha is more CPU-intensive.

You can get a 1-bit transparent image using a call to GraphicsConfiguration.createCompatibleImage(w,h, Transparency.BITMASK).

# **Text-Related Issues**

This section describes possible issues and crashes that are related to text rendering and describes tips to overcome such issues.

This section contains the following subsections:

- Application Crash During Text Rendering
- Differences in Text Appearance
- Metrics

### Application Crash During Text Rendering

If an application crashes during text rendering, first check the fatal error log file.

See Fatal Error Log for detailed information about this error log file. If the crash occurred in fontmanager.dll or if fontmanager is present in the stack, then the crash occurred in the font processing code. The following example shows typical native stack frames (excerpt from the full log file).

```
Stack: [0x008a0000,0x008f0000), sp=0x008ef52c, free space=317k
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code,
C=native code)
C [ntdll.dll+0x1888f]
С
  [ntdll.dll+0x18238]
C [ntdll.dll+0x11c76]
C [MSVCR71.dll+0x16b3]
C [MSVCR71.dll+0x16db]
С
  [fontmanager.dll+0x21f9a]
C [fontmanager.dll+0x22876]
C [fontmanager.dll+0x1de40]
С
 [fontmanager.dll+0x1da94]
С
 [fontmanager.dll+0x48abb]
j sun.font.FileFont.getGlyphImage(JI)J+0
j sun.font.FileFontStrike.getGlyphImagePtrs([I[JI)V+92
  sun.font.GlyphList.mapChars(Lsun/java2d/loops/FontInfo;I)Z+37
j
j sun.font.GlyphList.setFromString(Lsun/java2d/loops/FontInfo;Ljava/
lang/String;FF) Z+71
j sun.java2d.pipe.GlyphListPipe.drawString(Lsun/java2d/
SunGraphics2D;Ljava/lang/String;DD)V+148
j sun.java2d.SunGraphics2D.drawString(Ljava/lang/String;II)V+60
j FontCrasher.tryFont(Ljava/lang/String;)V+138
j
  FontCrasher.main([Ljava/lang/String;)V+20
```

v ~StubRoutines::call\_stub

In this case, a particular font is probably the problem. If so, then removing this font from the system will likely resolve the problem.



To identify the font file, execute the application with -Dsun.java2d.debugfonts=true. The font that is mentioned last is usually the one that is causing problems, as shown in the following example.

```
INFO: Registered file C:\WINDOWS\Fonts\WINGDING.TTF as font ** TrueType
Font: Family=Wingdings
Name=Wingdings style=0 fileName=C:\WINDOWS\Fonts\WINGDING.TTF rank=2
Aug 16, 2006 10:59:06 PM sun.font.FontManager initialiseDeferredFont
INFO: Opening deferred font file SYMBOL.TTF
Aug 16, 2006 10:59:06 PM sun.font.FontManager addToFontList
INFO: Add to Family Symbol, Font Symbol rank=2
Aug 16, 2006 10:59:06 PM sun.font.FontManager registerFontFile
INFO: Registered file C:\WINDOWS\Fonts\SYMBOL.TTF as font ** TrueType Font:
Family=Symbol
Name=Symbol style=0 fileName=C:\WINDOWS\Fonts\SYMBOL.TTF rank=2
Aug 16, 2006 10:59:06 PM sun.font.FontManager findFont2D
INFO: Search for font: Dialog
Aug 16, 2006 10:59:06 PM sun.font.FontManager initialiseDeferredFont
INFO: Opening deferred font file ARIALBD.TTF
Aug 16, 2006 10:59:06 PM sun.font.FontManager addToFontList
INFO: Add to Family Arial, Font Arial Bold rank=2
Aug 16, 2006 10:59:06 PM sun.font.FontManager registerFontFile
INFO: Registered file C:\WINDOWS\Fonts\ARIALBD.TTF as font ** TrueType Font:
Family=Arial
Name=Arial Bold style=1 fileName=C:\WINDOWS\Fonts\ARIALBD.TTF rank=2
Aug 16, 2006 10:59:06 PM sun.font.FontManager initialiseDeferredFont
INFO: Opening deferred font file WINGDING.TTF
Aug 16, 2006 10:59:06 PM sun.font.FontManager initialiseDeferredFont
INFO: Opening deferred font file SYMBOL.TTF
Aug 16, 2006 10:59:06 PM sun.font.FontManager findFont2D
INFO: Search for font: Dialog
Aug 16, 2006 10:59:06 PM sun.font.FontManager initialiseDeferredFont
INFO: Opening deferred font file ARIAL.TTF
Aug 16, 2006 10:59:06 PM sun.font.FontManager addToFontList
INFO: Add to Family Arial, Font Arial rank=2
Aug 16, 2006 10:59:06 PM sun.font.FontManager registerFontFile
INFO: Registered file C:\WINDOWS\Fonts\ARIAL.TTF as font ** TrueType Font:
Family=Arial
Name=Arial style=0 fileName=C:\WINDOWS\Fonts\ARIAL.TTF rank=2
Aug 16, 2006 10:59:06 PM sun.font.FontManager initialiseDeferredFont
INFO: Opening deferred font file WINGDING.TTF
Aug 16, 2006 10:59:06 PM sun.font.FontManager initialiseDeferredFont
INFO: Opening deferred font file SYMBOL.TTF
```

### Note:

In some cases, the font that is last mentioned might not be the problem. Font names are printed when they are first used and subsequent uses are not shown.

To verify that this particular font is causing the problem, you can temporarily remove it from your system. You can easily find the file name associated with this particular family name from the output.



Another verification approach is to use the Font2DTest tool to test fonts that you suspect. You can specify a particular font size, style, and rasterization mode. If the process of viewing a particular font with Font2DTest causes the JDK to crash, then it is very likely that it is the font that is causing the problems.

If you found a font causing the JDK to crash, it is very important to report this problem, including the particular font and the operating system in the Bugs Database. See Submit a Bug Report.

### Differences in Text Appearance

Java has its own font rasterizer, and you can expect some small differences between the appearance of text in a Java application and in a native application.

One of the typical sources of these differences is that the antialiasing settings can be different. In particular, a Swing application sometimes ignores the Linux desktop font antialiasing settings.

There are several likely reasons for this behavior:

- Over the remote X11 antialiasing is not enabled by default for performance reasons. See Font and Test questions in the Java 2D FAQ.
- CJK fonts that use embedded bitmaps may render using the bitmaps instead of subpixel text.
- Some variants of unsupported desktops do not report their font smoothing settings properly. For example, KDE is unsupported but should generally work; however, some problem seems to prevent JDK from picking up the setting.

The best way to ensure that the configuration is what you expect is to run Font2DTest, explicitly select the font used by the native application, and set other parameters as appropriate. Figure 12-1 is a sample screen from the *Font2DTest* tool.

💰 F	ont	2DT	est																																	×
File	Oş	otion	í												1										_											-
Font				Microsoft Sans Serif												Si	Size: 12							Fo	Font Transform: None							-				
Rang	je:		B	Basic Latin											-	St	Style:			Plain					Graphics Transform					n: N	None					-
Meth	dr	drawGivph/vector											1-	-										Text to use:					All Glyphs							
LCD	con	Irast	10	0	20	14	-	160	1	80	20	0	220	2	40	An	tialia	sin	g: 0	FF			•			Frac	tiona	al mi	etrics	81			0	FF		-
1				1		#	\$	%	8	•	(	)	+	+		-		1	0	1	2	3	4	5	6	7	8	9		1	<	=	>	?	0	
A	в	С	D	E	F	G	н	1	J	К	L	м	Ν	0	P	Q	R	s	Т	U	۷	W	х	Y	Z	1	x	1	^	-	2	a	ь	c	d	
е	f	9	h	i	j	k	1	m	n	0	P	q	٢	\$	<b>.</b> t	u	٧	w	×	У	z	(	1	1	~	Ă	Å	Ç	Ē	Ñ	Ŏ	0	á	à	å	
à	ã	å	ç	é	è	ê	ë	i	i	ī	ï	ñ	ó	ò	ő	ä	õ	ú	ù	û	ü	†		¢	٤	ş	•	1	ß	®.	٢	***	1	-	*	
Æ	ø	-	±	s	5	¥	μ	9	Σ	П	π	1	3	2	Ω	<b>36</b>	0	ż	i	-	4	f	.8	۵	۰	*		A	A	ō	Œ	08	-	-		
-	1		÷	0	Ŷ	Ŷ	1	ă	4	3	fi	fl	+	•	e.:		%0	Â	Ê	Á	Ë	Ê	ſ	1	Ĩ	1	Ó	Ô	ò	Ú	Û	Û	1	÷.,	*	
1	~	-	*		."		1	Ł	ł	Ś	ş	Z	ž	1	Ð	ð	Ý	ý	Þ	Þ	-	×		2	٥	1/2	14	94	F	Ğ	ğ	1	Ş	\$	¢	
ć	Ċ	ė	đ	-		Ă	ă	A	ą	Ď	d	Ð	Ę	ę	Ê	ě	Ĺ	1	Ľ	ſ	Ŀ	Ŀ	Ň	ń	Ň	ñ	ð	ő	Ŕ	ŕ	Ř	ř	ŝ	\$	τ	
5	Ť	٢	Ú	ů	Ū	ü	ź	ź	Ż	ż	Г	Θ	Φ	α	ō	c	σ	τ	φ	-	11	n	Pts	IJ	ij	'n	1		56	Ł	0	•	•	•		
Disp	layir	ig G	lyph	Cod	e 00	00 t	0 01	43						3	1-11 1		0			-		<u>.</u>										<u></u>				122

Figure 12-1 Sample Screen from Font2DTest Tool



### Tip:

You can input your own string by choosing **User Text** in the drop-down list labeled **Text to use**.

The size of the font in the Java language is always expressed with 72 dpi. A native OS can use a different screen dpi, and therefore an adjustment must be made. Matching Java font size can be calculated as Toolkit.getScreenResolution() divided by 72 multiplied by the size of the native font.

In all native Swing look and feel, such as the Windows look and feel or the GTK look and feel (for Oracle Solaris and Linux operating systems), Swing components perform this adjustment automatically, but if you are running *Font2DTest*, the text display area will always use 72 dpi.

On operating systems other than Windows, the general recommendation is to use TrueType fonts instead of Type1 fonts. The easiest way to figure out the type of font is to look at the file extension: extensions pfa and pfb indicate Type1 fonts, and ttf, ttc, and tte represent TrueType fonts.

### Metrics

If you find that text bounds are different from what you expect, then ensure that you are using the appropriate way to calculate them. For example, the height obtained from a FontMetrics is not specific to a particular piece of text, and the stringWidth indicates logical advance, which is not the same thing as *wide*. For more details, see the Font and Text questions in the Java 2D FAQ.

# Java 2D Printing

List of issues that can happen with Java 2D printing.

This section describes some issues that can happen with Java 2D printing and suggests causes and solutions.

Also, see the Printing questions in the Java 2D FAQ.

#### 1. JRE crashes during printing on Windows.

Cause: The JRE uses Windows printer drivers, and they might have problems.

Solution: Upgrade the Windows printer driver for the printer that is being used.

2. The printing seems to be successful, but the job does not print on Windows.

Cause: Some jobs fail to properly spool to the printer.

**Solution:** In the printer driver properties, disable Advanced Printing Options.

3. The print dialog box takes a long time to appear on Windows.

**Cause:** Applications might cause the JRE to probe all printers, including those that are disconnected.

**Solution:** Look for disconnected or unreachable network printers and remove them from the list of printers.

4. PrintJob.printDialog() shows no service found error on Oracle Solaris and Linux.



**Cause:** The cause is one of the following:

- The lpc utility is not in the /usr/sbin directory.
- The lpstat utility is not in the /usr/sbin directory.

Solution: Install  ${\tt lpc}$  and  ${\tt lpstat}$  in the standard location, as previously mentioned.

# 13 Swing

This chapter provides information and guidance on some specific procedures for troubleshooting some of the most common issues that might be found in the Java SE Swing API.

This chapter contains the following sections:

- General Debug Tips for Swing
- Specific Debug Tips for Swing

# General Debug Tips for Swing

Swing's painting infrastructure changed quite extensively in Java SE 6. If you notice painting artifacts specific in Java SE 6 or later releases, you can try turning off the new functionality. This can be done with the property swing.bufferPerWindow.

When you are debugging the Swing code which is executed while any menu is popped up, it is recommended to use the debugger remotely. Otherwise, the debugging process and the application execution block each other, and this prevents further work with the system. If that happens, the only action that can be taken is to kill the X server for Oracle Solaris and Linux. See Bug Database.

The following are some common Swing problems:

- Painting.
- Renderers.
- Updating models from wrong thread.
- Hangs.
- Responsiveness.
- Repainting issues.
- isOpaque usage.
- Startup: could be caused by small heap, loading unnecessary classes.

The following are some things to consider:

- Buffer-per-window feature.
- Native look-and-feel fidelity: Gnome vs Windows
- Footprint of Swing applications.
- JTable, JTree, and JList all use renderers.
- Make sure that custom renderers do as little as possible.
- Update models only from event dispatch thread. Otherwise the display will not reflect the state of the model.

The following identify bad renderers:



- Sluggish application, especially when scrolling.
- Use an optimizer to watch painting calls, look for calls to getTableCellTRendererComponent.

# Specific Debug Tips for Swing

The following topics describe problems in Swing and troubleshooting techniques:

- Incorrect Threading
- JComponent Children Overlap
- Display Update
- Model Change
- Add or Remove Components
- Opaque Override
- Permanent Changes to Graphics
- Custom Painting and Double Buffering
- Opaque Content Pane
- Renderer Call for Each Cell Performance
- Possible Leaks
- Mix Heavyweight and Lightweight Components
- Use Synth
- Track Activity on Event Dispatch Thread
- Specify Default Layout Manager
- Listener Object Dispatched to Incorrect Component
- Add a Component to Content Pane
- Drag and Drop Support
- One Parent for a Component
- JFileChooser Issues with Windows Shortcuts

### Incorrect Threading

Random exceptions and painting problems are usually the result of incorrect threading usage by Swing.

All access to Swing components, unless specifically noted in the javadoc, must be done on the event dispatch thread. This includes any models (TableModel, ListModel, and others) that are attached to Swing components.

The best way to check for bad usage of Swing is by using instrumented RepaintManager, as illustrated in the following example.

```
public class CheckThreadViolationRepaintManager extends RepaintManager
{
    // it is recommended to pass the complete check
```



```
private boolean completeCheck = true;
     public boolean isCompleteCheck() {
         return completeCheck;
     }
     public void setCompleteCheck(boolean completeCheck) {
         this.completeCheck = completeCheck;
     }
     public synchronized void addInvalidComponent(JComponent component) {
         checkThreadViolations(component);
         super.addInvalidComponent(component);
     }
     public void addDirtyRegion(JComponent component, int x, int y, int w,
int
h) {
         checkThreadViolations (component);
         super.addDirtyRegion(component, x, y, w, h);
     }
     private void checkThreadViolations(JComponent c) {
         if (!SwingUtilities.isEventDispatchThread() && (completeCheck ||
c.isShowing())) {
             Exception exception = new Exception();
             boolean repaint = false;
             boolean fromSwing = false;
             StackTraceElement[] stackTrace = exception.getStackTrace();
             for (StackTraceElement st : stackTrace) {
                 if (repaint &&
st.getClassName().startsWith("javax.swing.")) {
                     fromSwing = true;
                 }
                 if ("repaint".equals(st.getMethodName())) {
                     repaint = true;
                 ļ
             }
             if (repaint && !fromSwing) {
                 //no problems here, since repaint() is thread safe
                 return;
             }
             exception.printStackTrace();
         }
     }
}
```

### JComponent Children Overlap

Another possible source of painting problems can occur if you allow children of a <code>JComponent</code> to overlap.

In this case, the parent must override <code>isOptimizedDrawingEnabled</code> to return false. If you do not override <code>isOptimizedDrawingEnabled</code>, then components can randomly appear on top of others, depending upon which component repaint was invoked on.

## **Display Update**

Another source of painting problems can occur if you do not invoke repaint correctly when you need to update the display.

Changing a visible property of a Swing component, such as the font, will trigger a repaint or revalidate. If you are writing a custom component, then you must invoke repaint and possibly revalidate whenever the display or sizing information is updated. If you do not, the display will only update the next time someone triggers a repaint.

A good way to diagnose this is to resize the window. If the content appears after a resize, then that implies that the component did not invoke repaint or revalidate correctly.

### Model Change

Invoke repaint when you change a visible property of a Swing component, but you need not invoke repaint when your model changes.

If your model sends out the correct change notification, the JComponent will invoke repaint or revalidate as appropriate.

However, if you change your model but do not send out a notification, then a repaint event may not even work. In particular this will not work with JTree. The correct thing to do is to send the appropriate model notification. This can usually be diagnosed by resizing the window and noticing that the display did not update correctly.

### Add or Remove Components

When you add or remove components, you must manually invoke repaint or revalidate Swing and AWT.

### **Opaque Override**

Another possible area of painting problems is if a component does not override opaque.

Further, if you do not invoke implementation you must honor the opaque property, that is, if this component is opaque, you must completely fill in the background with a non-opaque color. If you do not honor the opaque property, then you will likely see visual artifacts.

The only way to check for this is to look for consistent visual artifacts when the component invokes repaint.

# Permanent Changes to Graphics

Do not make any permanent changes to a Graphics object that is passed to paint, paintComponent, or paintChildren.



Note:

If you override the graphics in a subclass, then you should not make permanent changes to the paint, paintComponent, or paintChildren passed in the Graphics object. For example, you should not alter the clip Rectangle or modify the transform. If you need to do these operations you may find it easier to create a new Graphics object from the passed in Graphics object and manipulate it instead.

If you ignore this restriction, then the result will be clipping or other weird visual artifacts.

# Custom Painting and Double Buffering

Although you can override paint and do custom painting in the override, you should instead override paintComponent.

The JComponent.paint method ensures that painting happens to the double buffer. If you override paint directly, then you may lose double buffering.

### **Opaque Content Pane**

Swing's painting architecture requires an opaque content pane.

The painting architecture of Swing requires an opaque JComponent to exist in the containment hierarchy above all other components. This is typically provided by using the content pane. If you replace the content pane, it is recommended that you make the content pane opaque by using setOpaque(true). Additionally, if the content pane overrides paintComponent, then it will need to completely fill in the background in an opaque color in paintComponent.

# Renderer Call for Each Cell Performance

Renderers are painted for each cell, so ensure that the renderer does as little as possible.

Any slowdown in the renderer is magnified across all cells. For example, if you repaint the visible region of a table with 50x20 visible cells, then there will be 1000 calls to the renderer.

### **Possible Leaks**

If the life cycle of your model is longer than that of a window with a component using the model, you must explicitly set the model of the Swing component to null.

If you do not set the model to null, your model will retain a reference to the Component, which will keep all components in the window from being garbage collected. Take a look at the following example.

```
TableModel myModel = ...;
JFrame frame = new JFrame();
frame.setContentPane(new JScrollPane(new JTable(myModel)));
frame.dispose();
```



If your application still holds a reference to myModel, then frame and all its children will still be reachable by way of the listener JTable installations on myModel. The solution is to invoke table.setModel(new DefaultTableModel()).

# Mix Heavyweight and Lightweight Components

Mixing heavyweight and lightweight components can work in certain scenarios, as long as the heavyweight component does not overlap with any existing Swing components.

For example, a heavyweight will not work in an internal frame, because when the user drags around the internal frame it will overlap with other internal frames. If you use heavyweights, then invoke the following methods:

- JPopupMenu.setDefaultLightWeightPopupEnabled(false)
- ToolTipManager.sharedInstance().setLightWeightPopupEnabled(f alse)

# Use Synth

Synth is an empty canvas.

To use Synth, you must either provide a complete XML file that configures the look and feel, or extend SynthLookAndFeel and provide your own SynthStyleFactory.

# Track Activity on Event Dispatch Thread

If a Swing application tries to do too much on the event dispatch thread, then the application will appear sluggish and unresponsive.

One way to detect this situation is to push a new EventQueue that can output logging information if an event takes too long to process. This approach is not perfect in that it has problems with focus events and modality, but it is good for ad-hoc testing.

# Specify Default Layout Manager

Problems can be caused by differing default layout manager classes on a Swing component.

For example, the default for the JPanel class is FlowLayout, but the default for the JFrame class is BorderLayout. This situation is easily fixed by specifying a LayoutManager.

# Listener Object Dispatched to Incorrect Component

MouseListener objects are dispatched to the deepest component that has MouseListener objects (or has enabled MouseEvent objects).

A ramification of this is if you attach a MouseListener to a component whose descendants have MouseListener objects, your MouseListener object will never get called.



This is easily reproduced with a composite component, like an editable JComboBox. Because a JComboBox has child components that have a MouseListener, a MouseListener attached to an editable JComboBox will never get notified.

If your MouseListener suddenly stops getting events, then it could be the result of a change in the application whereby a descendant component now has a MouseListener. A good way to check for this is to iterate over the descendants asking if they have any mouse listeners.

A similar scenario occurs with the KeyListener class. A KeyListener object is dispatched only to the focused component.

The JComboBox case is another example of this situation. In the editable JComboBox case the editor gets focus, not the JComboBox. As a result, a KeyListener attached to an editable JComboBox will never get notified.

# Add a Component to Content Pane

You must add a JFrame, JWindow, or JDialog component to the content pane.

A component added to a top-level Swing component must go to the content pane, but the add method (and a couple of other methods) on the JFrame, JWindow, and JDialog classes redirect to the content pane. In other words,

frame.getContentPane().add(component) is the same as
frame.add(component).

The following methods redirect to the content pane for you: add (and its variants), remove (and its variants), and setLayout.

This is purely a convenience, but can cause confusion. In particular, getChildren, getLayout, and various others do not redirect to the content pane.

This change affects LayoutManagers that only work with one component, such as GroupLayout and BoxLayout. For example, new GroupLayout (frame) will not work; instead, you must use GroupLayout (frame.getContentPane()).

# Drag and Drop Support

When using Swing you should use Swing's drag-and-drop support as provided by TransferHandler.

### One Parent for a Component

Remember that a component can only exist in one parent at a time.

Problems occur when you share menu items between menus. For example, JMenuItem is a component, and therefore can exist in only one menu at a time.

### JFileChooser Issues with Windows Shortcuts

The JFileChooser class does not support shortcuts on Windows OS (.Ink files).

Unlike the standard Windows file choosers, JFileChooser does not allow the user to follow Windows shortcuts when browsing the file system, because it does not show the correct path to the file.



To reproduce the problem, follow these steps:

- 1. Create a text file on the Desktop called, for example, MyFile.txt. Open the text file and type some text, for example: This is the contents of MyFile.txt.
- 2. Create a shortcut to the new text file in the following way: Drag the file with the right mouse button to another location on the Desktop and choose **Create Shortcut(s) here**.
- 3. Run the JfileChooser test application, browse the Desktop, select Shortcut to MyFile.txt and click Open.
- 4. The result file is *PathToDesktop*\Shortcut to MyFile.txt.lnk, but it should be *PathToDesktop*\MyFile.txt.
- 5. In addition, the contents of the result file in the text area shows the contents of the file shortcut to MyFile.txt.lnk, but the contents should be This is the contents of MyFile.txt, which was typed in step 1.



# 14 Internationalization

Information and guidance about troubleshooting issues that might be found in the area of internationalization support.

For detailed information, visit the Internationalization Overview.

This chapter describes troubleshooting techniques for internationalization and localization.

Troubleshoot Internationalization and Localization

# Troubleshoot Internationalization and Localization

Before troubleshooting, ensure that you understand the difference between *internationalization* and *localization*:

- Internationalization is the process of designing software so that it can be adapted (localized) to various languages and regions easily, in a cost-effective way, and without changes to the software. This process generally involves isolating the parts of a program that are dependent on language and culture. For example, the text of error messages are kept separate from the program source code because the messages must be translated during localization.
- Localization is the process of adapting a program for use in a specific locale. A locale is
  a geographic or political region that shares the same language and customs. Localization
  includes the translation of text such as user interface labels, error messages, and online
  help. It also includes the culture-specific formatting of data items such as monetary
  values, times, dates, and numbers.

The user interface libraries in the Java SE platform enable the development of rich interactive applications. The internationalization aspects include text input, text display, and user interface layout. The following descriptions show the relationship between internationalization and the functionality provided by the AWT, Java 2D, and Swing APIs:

- Text input is the process of entering new text into a document, whether by typing on a keyboard or through front-end software such as input methods, handwriting recognition, or speech input.
- Text display is a multistep process that includes selecting a font, arranging text into paragraphs and lines, selecting glyphs for characters or character sequences, and rendering these glyphs. Some writing systems require bidirectional text layout or complex character-to-glyph mappings. Text display is handled by the Java 2D graphics system and the Swing toolkit for lightweight user interface components and by AWT for peered user interface components.
- User interface layout needs to accommodate text expansion or shrinkage caused by localization, and match the direction of the user's writing system.



# 15 Java Sound

This chapter describes some issues that can arise with the Java sound technology and suggests causes and workarounds.

The following topic describes scenarios to troubleshoot Java sound problems.

Troubleshoot Java Sound Issues

# **Troubleshoot Java Sound Issues**

Troubleshoot Java sound issues such as system sound configuration, audio file format, audio format, and overrun and underrun conditions.

#### • System sound configuration:

Ensure that your audio system is correctly configured (sound card driver/DirectSound for Windows, ALSA for Linux, Audio Mixer for Oracle Solaris). In addition, ensure that your speakers are connected and that your sound card volume and mute state are adjusted to the appropriate value. To test your sound configuration, run any native sound application and play some sound through it.

On the Oracle Solaris and Linux operating systems, you might be unable to play sounds because an application (or sound daemon, such as esd or artsd) opens the audio device exclusively, thereby denying Java Sound access to the device.

#### • Audio file formats:

Java Sound supports a set of audio file formats, for example AU, AIF, and WAV. Most of the file formats are only containers and can contain audio data in various compressed audio formats. Java Sound file readers support some formats (uncompressed PCM, a-law, mu-law), but do not support ADPCM, MP3, and others.

Java Sound also supports plug-ins for file readers and writers through the service provider interface (SPI). You can use Sun, third-party, or your own plug-ins to read various audio files. In any case, you must handle the presence of the plug-in, for example, by distributing the required plug-ins with your application or by requiring plug-ins to be installed in the client Java environment.

#### • Audio formats:

Java Sound supports various audio formats, but their availability depends on the operating system. To use some audio format for recording or playing, the format must be supported by your system (sound card drivers). Use supported formats as much as possible: PCM; 8 or 16 bits; 8000, 11025, 22050, 44100 Hz. The formats are supported by most sound cards. Most sound cards support only PCM formats, and even if the driver supports mu-law, then it requires some modification to the software. If you need to play or record mu-law data, then the preferred way is to convert it to PCM format through a format converter.

See AudioSystem.getAudioInputStream documentation for details about format conversion.

• Overrun and underrun conditions:



Recorded data is kept in a DataLine buffer. If you did not read from the line for a long time, then an overrun condition will occur, and older data will be replaced with new data. This will produce artifacts in the recorded audio data.

A similar situation occurs with playing. If all data from the buffer has been played and no new data is written to the line, then an underrun condition will occur, and silence will be played until you write a new portion of audio data to the line.

The preferred way to record is to read data in a separate thread to prevent the possible influence of other tasks (for example, UI handling). If you use <code>SourceDataLine</code> for playing, then a separate thread for writing data into the line is also the preferred method to use. If you use <code>Clip</code> for playing, then the <code>Clip</code> implementation creates this type of thread itself.



# Part V Submit Bug Reports

The chapter Submit a Bug Report shows you how to submit a bug report. It includes suggestions about what to try before submitting a report and which data to collect for the report.



# 16 Submit a Bug Report

This chapter shows you how to submit a bug report. It includes suggestions about what to try before submitting a report and which data to collect for the report.

This chapter contains the following sections:

- Check for Fixes in Update Releases
- Prepare to Submit a Bug Report
- Collect Data for a Bug Report
- Report a Bug
- Collect Core Dumps

# Check for Fixes in Update Releases

Regularly scheduled updates to each release contain fixes for a set of critical bugs identified since the initial release of the platform.

When an update release becomes available, it becomes the default download at the Java SE Downloads page.

The download site includes a link to the release notes that list the bug fixes in the release. Each bug in the list is linked to the bug description in the bug database. The release notes also includes the list of fixes in previous update releases. If you encounter an issue, or suspect a bug, then, as an early step in the diagnosis, check the list of fixes that are available in the most recent update release.

Sometimes, it is not obvious if an issue is a duplicate of a bug that was already fixed.

# Prepare to Submit a Bug Report

The following is the recommended procedure to submit a bug report.

Before submitting a bug report, consider the following recommendations:

- Collect as much relevant data as possible. For example, generate a thread dump in the case of a deadlock, or locate the core file (where applicable) and hs\_err file in the case of a crash. In every case, it is important to document the environment and the actions performed just before the problem happened. See Collect Data for a Bug Report.
- Where applicable, try to restore the original state and reproduce the problem using the documented steps. This helps to determine if the problem is reproducible or an intermittent issue.
- If the issue is reproducible, try to narrow down the problem. In some cases, a bug can be demonstrated with a small standalone test case. Bugs that are demonstrated by small test cases will typically be easy to diagnose as compared to test cases that consist of a large complex application.



- Search the Java Bug Database to see if this issue or a similar issue was reported. If the issue has already been reported, then the bug report might have further information, such as the following:
  - If the issue was already fixed, then the release in which it was fixed is given.
  - A workaround for the issue.
  - Comments in the evaluation that explain, in further detail, the circumstances that cause the issue to happen.
- If you conclude that the issue was not already reported, then report it at Report a Bug or Request a Feature.

Before reporting an issue, verify that the environment where the problem happens is a supported configuration. See Oracle JDK 11 Certified System Configurations.

In addition to the system configurations, check the list of supported locales. See JDK 11 Supported Locales.

In the case of Oracle Solaris, check the recommended patch cluster for the operating system release to ensure that the recommended patches are installed.

# Collect Data for a Bug Report

The following sections list the commands or recommend a general procedure to obtain the data:

- Detailed Description of the Problem
- Hardware Details
- Operating System Details
- Java SE Version
- Command-Line Options
- Environment Variables
- Fatal Error Log
- Core and Crash Dump
- Logs and Traces

# Detailed Description of the Problem

When creating a problem description, try to include as much relevant information as possible.

Describe the application, the environment, and the most important events leading up to the time when the problem happened.

Report all troubleshooting steps and results that have already occurred.

Sometimes, the problem can be reproduced only in a complex application environment. In this case, the description, coupled with logs, core file, and other relevant information, might be the only way to diagnose the issue. In these situations, the description should indicate if the submitter is willing to run further diagnostics or run test binaries on the system where the issue occurs.



- If the problem is reproducible at will, then list the steps that are required to demonstrate the problem.
- If the problem can be demonstrated with a small test case, then include the test case and the commands to compile and execute the test case.
- If the test case or problem requires third-party code, such as a commercial or open source library or package, then provide then details about where and how to obtain the library.

## Hardware Details

The hardware details are stored in the error logs when a fatal error occurs.

Sometimes, a bug happens or can be reproduced only on certain hardware configurations. If a fatal error occurs, then the error log might contain the hardware details. If an error log is not available, then document in the bug report the number and the type of processors in the machine, the clock speed, and, where applicable and if known, some details on the features of that processor. For example, in the case of Intel processors, it might be relevant that hyper-threading is available.

# **Operating System Details**

The commands that you can use to get the operating system details.

On the Oracle Solaris operating system, the showrev -a command prints the operating system version and patch information.

On Linux, it is important to know which distribution and version is used. Sometimes the /etc/\*release file indicates the release information, but because components and packages can be upgraded independently, it is not always a reliable indication of the configuration. Therefore, in addition to the information from the \*release file, collect the following information:

- The kernel version. This can be obtained using the uname -a command.
- The glibc version. The rpm -q glibc command indicates the patch level of glibc.
- The thread library. There are two thread libraries for Linux, namely LinuxThreads and NPTL. The LinuxThreads library is used on 2.4, and earlier kernels and has fixed stack and floating stack variants. The Native POSIX Thread Library (NPTL) is used on the 2.6 kernel. Some Linux releases (such as RHEL3) include backports of NPTL to the 2.4 kernel. Use the command getconf GNU\_LIBPTHREAD\_VERSION to determine which thread library is used. If the getconf command returns an error to say that the variable does not exist, then it is likely that you are using an old kernel with the LinuxThreads library.

## Java SE Version

Obtain the Java SE version string with the java -version command.

Multiple versions of Java SE may be installed on the same machine. Therefore, ensure that you use the version of the java command used by the failing application. It is very likely to be different from the default java command included in a user's PATH environment variable.



# **Command-Line Options**

If the bug report does not include a fatal error log then, it is important to document the full command line and all its options. This includes any options that specify heap settings, for example, the -Xmx option, or any -XX options that specify HotSpot specific options.

If you can reproduce the issue at will, and you're able to read standard output (stdout) for the JVM, then you can add the -XX:+PrintCommandLineFlags option to obtain the full list of command-line options used by the application. This option will be active the next time the JVM is restarted.

You can also run the  $j\,{\tt cmd}$  command as follows to obtain the command-line options of a running VM:

jcmd <process ID for the Java process> VM.command line

In addition, you can change the flags of a running JVM through <code>jcmd</code>. See the <code>VM.set\_flag</code> command.

### **Environment Variables**

Sometimes problems arise due to environment variable settings. When creating the bug report, indicate the values of the following Java environment variables (if set).

- JAVA TOOL OPTIONS
- JAVA OPTIONS
- CLASSPATH
- JAVA COMPILER
- PATH
- USERNAME

#### Note:

You should obtain the values of environment variables from the context of the failing application. In addition, one or more configuration files may set the values of these environment variables for that failing application.

In addition, collect the following operating-system-specific environment variables.

- On Oracle Solaris and Linux operating systems, collect the values of the following environment variables:
  - LD LIBRARY PATH
  - LD\_PRELOAD
- On Windows, collect the values of the following environment variables:

– OS



- PROCESSOR\_IDENTIFIER
- \_ALT\_JAVA\_HOME\_DIR

# Fatal Error Log

When a fatal error occurs, an error log is created. See Fatal Error Log.

The error log contains information obtained at the time of the fatal error, such as version and environment information, details about the threads that provoked the crash, and so forth.

If the fatal error log is generated, then be sure to include it in the bug report or report it during a support call.

## Core and Crash Dump

If a core file or crash dump has been created because of the reported issue, include it with the bug, size permitting.

A Linux core file or a Windows crash dump contains the memory state of either an application or the operating system at the time the core or dump was created. Depending on a system's configuration, a core or crash dump may be created automatically when a crash happens. Consult with the system administrator to determine if a core file will be generate automatically and where.

In the case of a hung process, the procedure for generating a dump is described in Collect Core Dumps.

### Logs and Traces

Log or trace output can help to quickly determine the cause of a problem.

For example, in the case of a performance issue, the output of the -verbose:gc option can help in diagnosing the problem. This option enables output from the garbage collector.

In other cases, you can use Java Flight Recorder and JDK Mission Control to capture statistical information over the time period leading up to the problem.

In the case of a deadlock or a hung VM, the thread stacks can help diagnose the problem. Obtain thread stacks by pressing Control+\ on Oracle Solaris and Linux or Control+Break on Windows. Alternatively, use the Thread.dump to file option in the jcmd command.

In general, provide all relevant logs, traces, and other output in the bug report or during the support call.

# Report a Bug

Once you have concluded that your issue was not already reported and collected data about it, report it at Report a Bug or Request a Feature.

# **Collect Core Dumps**

A core dump or a crash dump is a memory snapshot of a process.



A core dump can be automatically created by the operating system when a fatal or unhandled error occurs. Alternatively, a core dump can be forced by using systemprovided command-line utilities. Sometimes, a core dump is useful when diagnosing a process that appears to be hung; the core dump may reveal information about the cause of the hang.

The following sections describe scenarios for collecting core dumps.

- Collect Core Dumps on Oracle Solaris
- Collect Core Dumps on Linux
- Reasons for Not Getting a Core File
- Collect Crash Dumps on Windows

### Collect Core Dumps on Oracle Solaris

By default, the core dump is created in the current working directory of the process. The name of the core dump file is core. The user can configure the location and name of the core dump using the core file administration utility, coreadm. This procedure is fully described in the man page for the coreadm utility.

The ulimit utility is used to get or set the limitations on the system resources available to the current shell and its descendants. Use the ulimit -c command to check or set the core file size limit. Ensure that the limit is set to unlimited; otherwise, the core file could be truncated.

#### Note:

ulimit is a Bash shell built-in command; on a C shell, use the limit command.

Ensure that any scripts that are used to launch the VM or your application do not disable core dump creation.

The gcore utility can be used to get a core image of running processes. This utility accepts a process ID (pid) of the process for which you want to force a core dump.

To get the list of Java processes running on the machine, you can use any of the following commands:

- ps -ef | grep java
- pgrep java
- jps

#### Note:

The jps command-line utility does not perform name matching (that is, looking for "java" in the process command name) and so it can list Java VM embedded processes as well as the Java processes.



The following are two methods to collect core dumps on Oracle Solaris.

#### ShowMessageBoxOnError option on Oracle Solaris:

A Java process can be started with the -XX:+ShowMessageBoxOnError command-line option. When a fatal error occurs, the process prints a message to standard error and waits for a yes or no response from standard input.

#### Suspend a process with the truss utility:

In situations where it is not possible to specify the -XX:+ShowMessageBoxOnError option, you might be able to use the truss utility. This Oracle Solaris operating system utility is used to trace system calls and signals. You can use this utility to suspend the process when it reaches a specific function or system call.

The command in the following example shows how to use the truss utility to suspend a process when the exit system call is executed (in other words, the process is about to exit).

```
$ truss -t \!all -s \!all -T exit -p pid
```

When the process calls exit, it will be suspended. At this point, you can attach the debugger to the process or call gcore to force a core dump.

### Collect Core Dumps on Linux

By default, the core dump is created in the current working directory of the process. The name of the core dump file is core.pid, where pid is the process ID of the crashed Java process.

Not all systems are configured to allow the creation of core files. The ulimit utility is used to get or set the limitations on the system resources available to the current shell and its descendants. Use the ulimit -c command to check or set the core file size limit. Ensure that the limit is set to unlimited; otherwise, the core file could be truncated or not produced.

#### Note:

ulimit is a Bash shell built-in command; on a C shell, use the limit command.

Ensure that any scripts that are used to launch the VM or your application do not disable core dump creation.

You can use the gcore command in the gdb (GNU debugger) interface to get a core image of a running process. This utility accepts the pid of the process for which you want to force the core dump.

To get the list of Java processes running on the machine, you can use any of the following commands:

- jcmd
- ps -ef | grep java
- pgrep java



You can use the ShowMessageBoxOnError option to collect core dumps on Linux. Start a Java process with the -XX:+ShowMessageBoxOnError command-line option. When a fatal error occurs, the process prints a message to standard error and waits for a yes or no response from standard input.

# Reasons for Not Getting a Core File

The following is a list of reasons why a core file might not be generated on Linux:

- The application user does not have permission to write in the current working directory of the process.
- The application user has write permission on the current working directory, but there is already a file named core that has read-only permission.
- The current directory does not have enough space.
- The current directory has a subdirectory named core.
- The current working directory is remote. It might be mapped to a Network File System (NFS), and NFS failed at the time the core dump was about to be created.
- Oracle Solaris operating system only: The coreadm tool has been used to configure the directory and name of the core file, but one or more of the previous reasons apply to the configured directory.
- The process is running a setuid program, and therefore the operating system will not dump the core unless it is configured explicitly.
- Java specific: If the process received SIGSEGV or SIGILL but did not produce a core file, it is possible that the process handled it. For example, HotSpot VM uses the SIGSEGV signal for legitimate purposes, such as throwing NullPointerException, deoptimization, and so forth. The signal is unhandled by the Java VM only if the current instruction (PC) falls outside the Java VM generated code. These are the only cases in which HotSpot dumps the core.
- Java specific: The JNI Invocation API was used to create the VM. The standard Java launcher was not used. The custom Java launcher program handled the signal by consuming it and produced the log entry silently. This situation has occurred with certain application servers and web servers. These Java VM embedding programs transparently attempt to restart (fail over) the system after an abnormal termination. In this case, the fact that a core dump is not produced is a feature and not a bug.

# Collect Crash Dumps on Windows

In the Windows operating system, there are three types of crash dumps: Dr. Watson log file, user minidump, and Dr. Watson full dump.

- Dr. Watson log file, which is a text error log file that includes faulting stack trace and a few other details.
- User minidump, which is considered a partial core dump. It is not a complete core dump, because it does not contain all the useful memory pages of the process.
- Dr. Watson full dump, which is equivalent to a UNIX core dump. This dump contains most memory pages of the process except for code pages.



When an unexpected exception occurs on Windows, the action taken depends on two values in the following registry key:

\\HKEY LOCAL MACHINE\Software\Microsoft\Windows NT\CurrentVersion\AeDebug

The two values are named Debugger and Auto. The Auto value indicates if the debugger specified in the value of the Debugger entry starts automatically when an application error occurs.

- A value of 0 for Auto means that the system displays a message box notifying the user when an application error occurs.
- A value of 1 for Auto means that the debugger starts automatically.

The value of Debugger is the debugger command that is to be used to debug program errors.

When a program error occurs, Windows examines the Auto value, and if the value is 0, then it executes the command in the Debugger value. If the value for Debugger is a valid command, then a message box is created with two buttons: **OK** and **Cancel**. If the user clicks **OK**, then the program is terminated. If the user clicks **Cancel**, then the specified debugger is started. If the value for the Auto entry is set to 1 and the value for the Debugger entry specifies the command for a valid debugger, then the system automatically starts the debugger and does not generate a message box.

The following are two ways to collect crash dump on Windows.

#### Configure Dr.Watson:

The Dr. Watson debugger is used to create crash dump files. By default, the Dr. Watson debugger (drwtsn32.exe) is installed in the Windows system folder (%SystemRoot% \System32).

To install Dr. Watson as the postmortem debugger, run the following command:

drwtsn32 -i

To configure the name and location of crash dump files, run drwtsn32 without any options.

In the Dr. Watson GUI window, ensure that the **Create Crash Dump File** check box is selected and that the crash dump file path and log file path are configured in their respective text fields.

Dr. Watson can be configured to create a full dump using the registry. The registry key is shown in the following example.

```
System Key: [HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\DrWatson]
Entry Name: CreateCrashDump
Value: (0 = disabled, 1 = enabled)
```



#### Note:

If the application handles the exception, then the registry-configured debugger is not invoked. In that case, it might be appropriate to use the – XX:+ShowMessageBoxOnError command-line option to force the process to wait for user intervention on fatal error conditions.

#### • Force a crash dump:

The userdump command-line utility can be used to force a Dr. Watson dump of a running process. The userdump utility does not ship with Windows. It is released as a component of the OEM Support Tools package.

An alternative way to force a crash dump is to use the windbg debugger. The main advantage of using windbg is that it can attach to a process in a non-invasive manner (that is, read-only). Usually, Windows terminates a process after a crash dump is obtained, but with the noninvasive attach, it is possible to obtain a crash dump and let the process continue. To attach the debugger check box requires selecting the **Attach to Process** option and the **Noninvasive** check box.

When the debugger is attached, a crash dump can be obtained using the command shown in the following example.

.dump /f crash.dmp

The windbg debugger is included in the Debugging Tools for Windows download.

An additional utility in this download is the dumpchk.exe utility, which can verify that a memory dump file was created correctly.

Both userdump.exe and windbg require the pid of the process. The userdump -p command lists the process and program for all processes. This is useful if you know that the application is started with the java.exe launcher. However, if a custom launcher is used (embedded VM), then it might be difficult to recognize the process. In that case, you can use the jps command-line utility because it lists the PIDs of the Java processes only.

You can also use the -XX:+ShowMessageBoxOnError command-line option on Windows. When a fatal error occurs, the process shows a message box and waits for a yes or no response from the user.

Before clicking **Yes** or **No**, you can use the userdump.exe utility to generate the Dr. Watson dump for the Java process. This utility can also be used in cases when the process appears to be hung.



# Part VI Appendices

This part contains the following topics.

• Fatal Error Log

Describes fatal error log contents and location

• Java 2D Properties

Describes properties that are useful in troubleshooting issues with Java 2D

• Environment Variables and System Properties

Describes environment variables and system properties that are useful when troubleshooting issues with Java HotSpot Server VM

Command-Line Options

Describes command-line options that are useful when diagnosing issues with Java HotSpot Server VM

• Summary of Tools in This Release

Provides a summary of the tools available in the current and previous releases of the JDK.

# A Fatal Error Log

The fatal error log is created when a fatal error occurs. It contains information and the state obtained at the time of the fatal error.

#### Note:

The format of this file can change slightly in update releases.

This appendix contains the following sections:

- Location of Fatal Error Log
- Description of Fatal Error Log
- Header Format
- Thread Section Format
- Process Section Format
- System Section Format

# Location of Fatal Error Log

To specify where the log file will be created, use the product flag -XX:ErrorFile=file, where file represents the full path for the log file location.

The substring %% in the *file* variable is converted to %, and the substring %p is converted to the PID of the process.

In the following example, the error log file will be written to the directory /var/log/java and will be named java errorpid.log:

java -XX:ErrorFile=/var/log/java/java\_error%p.log

If the -XX:ErrorFile=file flag is not specified, then the default log file name is hs err *pid*.log, where *pid* is the PID of the process.

In addition, if the -XX: ErrorFile=file flag is not specified, the system attempts to create the file in the working directory of the process. In the event that the file cannot be created in the working directory (insufficient space, permission problem, or other issue), the file is created in the temporary directory for the operating system. On the Oracle Solaris and Linux operating systems, the temporary directory is / tmp. On the Windows, the temporary directory is specified by the value of the TMP environment variable. If that environment variable is not defined, then the value of the TEMP environment variable is used.



# **Description of Fatal Error Log**

The error log contains information obtained at the time of the fatal error, including the following information, where possible:

- The operating exception or signal that provoked the fatal error
- Version and configuration information
- Details about the thread that provoked the fatal error and the thread's stack trace
- List of running threads and their states
- Summary information about the heap
- List of native libraries loaded
- Command-line arguments
- Environment variables
- Details about the operating system and CPU

#### Note:

In some cases only a subset of this information is output to the error log. This can happen when a fatal error is of such severity that the error handler is unable to recover and report all the details.

The error log is a text file consisting of the following sections:

- A header that provides a brief description of the crash. See Header Format.
- A section with thread information. See Thread Section Format.
- A section with process information. See Process Section Format.
- A section with system information. See System Section Format.

#### Note:

The format of the fatal error log described here is based on Java SE 6. The format might be different with other releases.

# Header Format

The header section at the beginning of every fatal error log file contains a brief description of the problem.

The header is also printed to standard output and may show up in the application's output log.



The header includes a link to the HotSpot Virtual Machine Error Reporting Page, where the user can submit a bug report.

```
#
# A fatal error has been detected by the Java Runtime Environment:
#
#
  SIGSEGV (0xb) at pc=0x00007f0f159f857d, pid=18240, tid=18245
#
# JRE version: Java(TM) SE Runtime Environment (9.0+167) (build 9-ea+167)
# Java VM: Java HotSpot(TM) 64-Bit Server VM (9-ea+167, mixed mode, tiered,
compressed oops, g1 gc, linux-amd64)
# Problematic frame:
# C [libMyApp.so+0x57d] Java MyApp readData+0x11
# Core dump will be written. Default location: /cores/core.18240)
#
# If you would like to submit a bug report, please visit:
  http://bugreport.java.com/bugreport/crash.jsp
# The crash happened outside the Java Virtual Machine in native code.
# See problematic frame for where to report the bug.
#
```

The example shows that the VM crashed on an unexpected signal.

The next line describes the signal type, program counter (pc) that caused the signal, process ID, and thread ID, as shown in the following example.

The following line and table describes the signal type, program counter (pc) that caused the signal, process ID, and thread ID.

# SIGSEGV (0xb) at pc=0x00007f0f159f857d, pid=18240, tid=18245

Table A-1 Line Description

Line Component	Description
SIGSEGV	Signal name
(0xb)	Signal number
pc=0x00007f0f159f857d	Program counter (instruction pointer)
pid=18240	Process ID
tid=18245	Thread ID

The next line contains the VM version (client VM or server VM), an indication of whether the application was run in mixed or interpreted mode, and an indication of whether class file sharing was enabled, as shown in the following line.

# Java VM: Java HotSpot(TM) 64-Bit Server VM (9-ea+167, mixed mode, tiered, compressed oops, g1 gc, linux-amd64)



The next information is the function frame that caused the crash, as shown in the following example.

```
# Problematic frame:
# C [libMyApp.so+0x57d] Java MyApp readData+0x11
```

Table A-2 Line Description

Line Component	Description
C	Frame type
[libMyApp.so+0x57d] Java_MyApp_readData+0x11	Same as program counter, but represented as library name and offset. For position- independent libraries (JVM and most shared libraries), it is possible to inspect the instructions that caused the crash without a debugger or core file by using a disassembler to dump instructions near the offset.

In this example, the "C" frame type indicates a native C frame. Table A-3 shows the possible frame types.

Table A-3 Frame Types

Frame Type	Description
С	Native C frame
j	Interpreted Java frame
V	VM frame
v	VM-generated stub frame
J	Other frame types, including compiled Java frames

Internal errors will cause the VM error handler to generate a similar error dump. However, the header format is different. Examples of internal errors are guarantee() failure, assertion failure, ShouldNotReachHere(), and so forth. The following example shows the header format for an internal error.

```
#
# An unexpected error has been detected by HotSpot Virtual Machine:
#
# Internal Error (4F533F4C494E55583F491418160E43505000F5), pid=10226,
tid=16384
#
# Java VM: Java HotSpot(TM) Client VM (1.6.0-rc-b63 mixed mode)
```

In the above header, there is no signal name or signal number. Instead the second line now contains Internal Error and a long hexadecimal string. This hexadecimal string encodes the source module and line number where the error was detected. In general this "error string" is useful only to engineers working on the HotSpot Virtual Machine.



The error string encodes a line number and therefore it changes with each code change and release. A crash with a given error string in one release (for example, 1.6.0) might not correspond to the same crash in an update release (for example, 1.6.0\_01), even if the strings match.

#### Note:

Do not assume that a workaround or solution that worked in one situation associated with a given error string will work in another situation associated with that same error string. Note the following facts:

- Errors with the same root cause might have different error strings.
- Errors with the same error string might have completely different root causes.

Therefore, the error string should not be used as the sole criterion when troubleshooting bugs.

# **Thread Section Format**

Information about the thread that crashed.

If multiple threads crash at the same time, then only one thread is printed.

#### **Thread Information**

The first part of the thread section shows the thread that caused the fatal error, as shown in the following example.

```
Current thread (0x00007f102c013000): JavaThread "main" [_thread_in_native, id=18245, stack(0x00007f10345c0000,0x00007f10346c0000)]
```

Table A-4	Thread	Information
-----------	--------	-------------

Thread Component	Description
0x00007f102c013000	Thread Pointer
JavaThread	Thread Type
main	Thread Name
_thread_in_native	Thread State
id=18245	Thread ID
stack(0x00007f10345c0000,0x00007f10346c0000)	Stack

The thread pointer is the pointer to the Java VM internal thread structure. It is generally of no interest unless you are debugging a live Java VM or core file.

The following list shows possible thread types.

- JavaThread
- VMThread
- CompilerThread
- GCTaskThread



- WatcherThread
- ConcurrentMarkSweepThread

Table A-5 shows the important thread states.

Table A-5 Thread States

Thread State	Description
_thread_uninitialized	Thread is not created. This occurs only in the case of memory corruption.
_thread_new	Thread was created, but it has not yet started.
_thread_in_native	Thread is running native code. The error is probably a bug in the native code.
_thread_in_vm	Thread is running VM code.
_thread_in_Java	Thread is running either interpreted or compiled Java code.
_thread_blocked	Thread is blocked.
trans	If any of the previous states is followed by the string _trans, then that means that the thread is changing to a different state.

The thread ID in the output is the native thread identifier.

If a Java thread is a daemon thread, then the string daemon is printed before the thread state.

#### **Signal Information**

The next information in the error log describes the unexpected signal that caused the VM to terminate. On a Windows system the output appears as shown in the following example.

```
siginfo: ExceptionCode=0xc0000005, reading address 0xd8ffecf1
```

In the above example, the exception code is 0xc0000005 (ACCESS\_VIOLATION), and the exception occurred when the thread attempted to read address 0xd8ffecf1.

On Oracle Solaris and Linux operating systems the signal number (si\_signo) and signal code (si code) are used to identify the exception, as follows:

```
siginfo: si_signo: 11 (SIGSEGV), si_code: 1 (SEGV_MAPERR), si_addr:
0x0000000000000000
```

#### **Register Context**

The next information in the error log shows the register context at the time of the fatal error. The exact format of this output is processor-dependent. The following example shows output for the Intel(R) Xeon(R) processor.

```
Registers:
RAX=0x000000000000000, RBX=0x00007f0f17aff3b0,
RCX=0x000000000000001, RDX=0x00007f1033880358
RSP=0x00007f10346be930, RBP=0x00007f10346be930,
```



```
RSI=0x00007f10346be9a0, RDI=0x00007f102c013218

R8 =0x00007f0f17aff3b0, R9 =0x0000000000000, R10=0x00007f1011bb1de9,

R11=0x00000000000000, R13=0x00007f0f17aff3b0, R14=0x00007f10346be9a8,

R15=0x00007f102c013000

RIP=0x00007f0f159f857d, EFLAGS=0x00000000010283,

CSGSFS=0x00000000000033, ERR=0x000000000004
```

The register values might be useful when combined with instructions, as described below.

#### **Machine Instructions**

After the register values, the following example shows the error log that contains the top of stack followed by 32 bytes of instructions (opcodes) near the program counter (PC) when the system crashed. These opcodes can be decoded with a disassembler to produce the instructions around the location of the crash. *Note:* IA32 and AMD64 instructions are variable in length, and so it is not always possible to reliably decode instructions before the crash PC.

```
Top of Stack: (sp=0x00007f10346be930)

0x00007f10346be930: 00007f10346be990 00007f1011bb1e15

0x00007f10346be940: 00007f1011bb1b33 00007f10346be948

0x00007f10346be950: 00007f0f17aff3b0 00007f10346be9a8

0x00007f10346be960: 00007f0f17aff5a0 0000000000000

Instructions: (pc=0x00007f0f159f857d)

0x00007f0f159f855d: 3d e6 08 20 00 ff e0 0f 1f 40 00 5d c3 90 90 55

0x00007f0f159f855d: 48 89 e5 48 89 7d f8 48 89 75 f0 b8 00 00 00

0x00007f0f159f857d: 8b 00 5d c3 90 90 90 90 90 90 90 90 90 90

0x00007f0f159f858d: 90 90 90 55 48 89 e5 53 48 83 ec 08 48 8b 05 88
```

#### **Thread Stack**

Where possible, the next output in the error log is the thread stack, as shown in the following example. This includes the addresses of the base and the top of the stack, the current stack pointer, and the amount of unused stack available to the thread. This is followed, where possible, by the stack frames, and up to 100 frames are printed. For C/C++ frames, the library name may also be printed. *Note:* In some fatal error conditions, the stack may be corrupt, and this detail may not be available.

```
Stack: [0x00007f10345c0000,0x00007f10346c0000], sp=0x00007f10346be930,
free space=1018k
Native frames: (J=compiled Java code, A=aot compiled Java code,
j=interpreted, Vv=VM code, C=native code)
C [libMyApp.so+0x57d] Java MyApp readData+0x11
i
  MyApp.readData()I+0
j MyApp.main([Ljava/lang/String;)V+15
v ~StubRoutines::call stub
V [libjvm.so+0x839eea] JavaCalls::call helper(JavaValue*, methodHandle
const&, JavaCallArguments*, Thread*)+0x47a
V [libjvm.so+0x896fcf] jni invoke static(JNIEnv *, JavaValue*, jobject*,
JNICallType, jmethodID*, JNI ArgumentPusher*, Thread*)
[clone .isra.90]+0x21f
V [libjvm.so+0x8a7fle] jni CallStaticVoidMethod+0x14e
C [libjli.so+0x4142] JavaMain+0x812
```

```
C [libpthread.so.0+0x7e9a] start_thread+0xda
```

```
Java frames: (J=compiled Java code, j=interpreted, Vv=VM code)
```

```
j MyApp.readData()I+0
```

- j MyApp.main([Ljava/lang/String;)V+15
- v ~StubRoutines::call\_stub

The log contains two thread stacks.

• The first thread stack is Native frames, which prints the native thread showing all function calls. However, this thread stack does not take into account the Java methods that are inlined by the runtime compiler; if methods are inlined, then they appear to be part of the parent's stack frame.

The information in the thread stack for native frames provides important information about the cause of the crash. By analyzing the libraries in the list from the top down, you can generally determine which library might have caused the problem and report it to the appropriate organization responsible for that library.

• The second thread stack is Java frames, which prints the Java frames including the inlined methods, skipping the native frames. Depending on the crash, it might not be possible to print the native thread stack, but it might be possible to print the Java frames.

#### **Further Details**

If the error occurred in the VM thread or in a compiler thread, then further details may be seen from the following example. For example, in the case of the VM thread, the VM operation is printed if the VM thread is executing a VM operation at the time of the fatal error. In the following output example, the compiler thread caused the fatal error. The task is a compiler task, and the HotSpot Client VM is the compiling method hs101t004Thread.ackermann.

```
Current CompileTask:
HotSpot Client Compiler:754 b
nsk.jvmti.scenarios.hotswap.HS101.hs101t004Thread.ackermann(IJ)J (42
bytes)
```

For the HotSpot Server VM, the output for the compiler task is slightly different but will also include the full class name and method.

# **Process Section Format**

The process section is printed after the thread section.

It contains information about the whole process, including the thread list and memory usage of the process.

#### **Thread List**

The thread list includes the threads that the VM is aware of, as shown in the following example.

```
=>0x0805ac88 JavaThread "main" [_thread_in_native, id=21139,
stack(0x00007f10345c0000,0x00007f10346c0000)]
```



Thread Component	Description
=>	Current Thread
0x0805ac88	Thread Pointer
JavaThread	Thread Type
main	Thread Name
_thread_in_native	Thread State
id=21139	Thread ID
stack(0x00007f10345c0000,0x00007f10346c0000)	Stack

Table A-6	Thread L	ist Description
-----------	----------	-----------------

This includes all Java threads and some VM internal threads, but does not include any native threads created by the user application that have not attached to the VM, as shown in the following example.

```
Other Threads:

0x00007f102c43f000 VMThread "VM Thread" [stack:

0x00007f0f175ff000,0x00007f0f176ff000] [id=18296]

0x00007f012c54b000 WatcherThread [stack:

0x00007f0f15bfb000,0x00007f0f15cfb000] [id=18338]
```

The thread type and thread state are described in Thread Section Format.

#### VM State

The next information is the VM state, which indicates the overall state of the virtual machine. Table A-7 describes the general states.

General VM State	Description
not at a safepoint	Normal execution.
at safepoint	All threads are blocked in the VM waiting for a special VM operation to complete.
synchronizing	A special VM operation is required, and the VM is waiting for all threads in the VM to block.

#### Table A-7 VM States



The VM state output is a single line in the error log, as follows:

VM state:not at safepoint (normal execution)

#### **Mutexes and Monitors**

The next information in the error log is a list of mutexes and monitors that are currently owned by a thread, as shown in the following example. These mutexes are VM internal locks rather than monitors associated with Java objects. The following is an example to show how the output might look when a crash happens when VM locks are held. For each lock, the log contains the name of the lock, its owner, and the addresses of a VM internal mutex structure and its OS lock. In general, this information is useful only to those who are very familiar with the HotSpot VM. The owner thread can be cross-referenced to the thread list.

```
VM Mutex/Monitor currently owned by a thread:
([mutex/lock_event])[0x007357b0/0x0000031c] Threads_lock - owner
thread: 0x00996318
[0x00735978/0x000002e0] Heap lock - owner thread: 0x00736218
```

#### **Heap Summary**

The next information is a summary of the heap, as shown in the following example. The output depends on the garbage collection (GC) configuration. In this example, the serial collector is used, class data sharing is disabled, and the tenured generation is empty. This probably indicates that the fatal error occurred early or during startup, and a GC has not yet promoted any objects into the tenured generation.

```
Heap
def new generation total 576K, used 161K [0x46570000, 0x46610000,
0x46a50000)
  eden space 512K, 31% used [0x46570000, 0x46598768, 0x465f0000)
  from space 64K, 0% used [0x465f0000, 0x465f0000, 0x46600000)
  to space 64K, 0% used [0x46600000, 0x46600000, 0x46610000)
  tenured generation total 1408K, used 0K [0x46a50000, 0x466b0000,
0x4a570000)
   the space 1408K, 0% used [0x46a50000, 0x46a50000, 0x46a50200,
0x46bb0000)
  compacting perm gen total 8192K, used 1319K [0x4a570000, 0x4ad70000,
0x4e570000)
   the space 8192K, 16% used [0x4a570000, 0x4a6b9d48, 0x4a6b9e00,
0x4ad70000]
No shared spaces configured.
```

#### **Memory Map**

The next information in the log is a list of virtual memory regions at the time of the crash. This list can be long if the application is large. The memory map can be very useful when debugging some crashes, because it can tell you which libraries are actually being used, their location in memory, as well as the location of the heap, stack, and guard pages.

The format of the memory map is operating system-specific. On the Oracle Solaris operating system, the base address and library name are printed. On the Linux



system, the process memory map (/proc/pid/maps) is printed. On the Windows system, the base and end addresses of each library are printed. The following example shows the output generated on Linux/x86.

Note: Most of the lines were omitted from the example for the sake of brevity.

```
Dynamic libraries:
00400000-00401000 r-xp 00000000 00:47 1374716350
export/java re/jdk/9/ea/167/binaries/linux-x64/bin/java
00601000-00602000 rw-p 00001000 00:47 1374716350
export/java re/jdk/9/ea/167/binaries/linux-x64/bin/java
016c6000-016e7000 rw-p 00000000 00:00 0
[heap]
82000000-102000000 rw-p 00000000 00:00 0
102000000-80000000 ---p 00000000 00:00 0
40014000-40015000 r--p 00000000 00:00 0
Lines omitted
7f0f159f8000-7f0f159f9000 r-xp 00000000 08:11 116808980
export/users/dh198349/tests/hs-err/libMyApp.so
7f0f159f9000-7f0f15bf8000 ---p 00001000 08:11 116808980
export/users/dh198349/tests/hs-err/libMyApp.so
7f0f15bf8000-7f0f15bf9000 r--p 00000000 08:11 116808980
export/users/dh198349/tests/hs-err/libMyApp.so
7f0f15bf9000-7f0f15bfa000 rw-p 00001000 08:11 116808980
export/users/dh198349/tests/hs-err/libMyApp.so
Lines omitted.
7f0f15dfc000-7f0f15e00000 ---p 00000000 00:00 0
7f0f15e00000-7f0f15efd000 rw-p 00000000 00:00 0
7f0f15efd000-7f0f15f13000 r-xp 00000000 00:47 1374714565
export/java re/jdk/9/ea/167/binaries/linux-x64/lib/libnet.so
7f0f15f13000-7f0f16113000 ---p 00016000 00:47 1374714565
export/java re/jdk/9/ea/167/binaries/linux-x64/lib/libnet.so
7f0f16113000-7f0f16114000 rw-p 00016000 00:47 1374714565
export/java re/jdk/9/ea/167/binaries/linux-x64/lib/libnet.so
7f0f16114000-7f0f16124000 r-xp 00000000 00:47 1374714619
export/java re/jdk/9/ea/167/binaries/linux-x64/lib/libnio.so
Lines omitted.
7f0f17032000-7f0f17036000 ---p 00000000 00:00 0
7f0f17036000-7f0f17133000 rw-p 00000000 00:00 0
7f0f17133000-7f0f173fc000 r--p 00000000 08:02
2102853
                           /usr/lib/locale/locale-archive
7f0f173fc000-7f0f17400000 ---p 00000000 00:00 0
Lines omtted.
```

#### The following is a format of memory map in the error log.

40049000-4035c000 r-xp 00000000 03:05 824473 /jdk1.5/jre/lib/i386/client/ libjvm.so



Memory Map Component	Description
40049000-4035c000	Memory region
r-xp	Permission: • read • write • execute • private • share
0000000	File offset
03:05	Major ID and minor ID of the device where the file is located (that is /dev/hda5)
824473	An inode number
/jdk1.5/jre/lib/i386/client/libjvm.so	File name

Table A-8	Memory	Map Format	Description
-----------	--------	------------	-------------

The example shows the memory map output and each library has two virtual memory regions: one for code and one for data. The permission for the code segment is marked with r-xp (readable, executable, private), and the permission for the data segment is rw-p (readable, writable, private).

The Java heap is already included in the heap summary earlier in the output, but it can be useful to verify that the actual memory regions reserved for the heap match the values in the heap summary and that the attributes are set to rwxp.

Thread stacks usually show up in the memory map as two back-to-back regions, one with permission --p (guard page) and one with permission rwxp (actual stack space). In addition, it is useful to know the guard page size or stack size. For example, in this memory map, the stack is located from 4127b000 to 412fb000.

On a Windows system, the memory map output is the load and end address of each loaded module, as shown in the following example.

Dynamic libraries:	
0x00400000 - 0x0040c000 c:\jdk6\bin\java.exe	
0x77f50000 - 0x77ff7000 C:\WINDOWS\System32\ntdll.dll	
0x77e60000 - 0x77f46000 C:\WINDOWS\system32\kernel32.dl	1
0x77dd0000 - 0x77e5d000 C:\WINDOWS\system32\ADVAPI32.dl	1
0x78000000 - 0x78087000 C:\WINDOWS\system32\RPCRT4.dll	
0x77c10000 - 0x77c63000 C:\WINDOWS\system32\MSVCRT.dll	
0x08000000 - 0x08183000 c:\jdk6\jre\bin\client\jvm.dll	
0x77d40000 - 0x77dcc000 C:\WINDOWS\system32\USER32.dll	
0x7e090000 - 0x7e0d1000 C:\WINDOWS\system32\GDI32.dll	
0x76b40000 - 0x76b6c000 C:\WINDOWS\System32\WINMM.dll	
0x6d2f0000 - 0x6d2f8000 c:\jdk6\jre\bin\hpi.dll	
0x76bf0000 - 0x76bfb000 C:\WINDOWS\System32\PSAPI.DLL	
0x6d680000 - 0x6d68c000 c:\jdk6\jre\bin\verify.dll	
0x6d370000 - 0x6d38d000 c:\jdk6\jre\bin\java.dll	
0x6d6a0000 - 0x6d6af000 c:\jdk6\jre\bin\zip.dll	
0x10000000 - 0x10032000 C:\bugs\crash2\App.dll	



#### VM Arguments and Environment Variables

The next information in the error log is a list of VM arguments, followed by a list of environment variables, as shown in the following example.

```
VM Arguments:
jvm_args:
java_command: MyApp
java_class_path (initial): .
Launcher Type: SUN_STANDARD
Logging:
Log output configuration:
#0: stdout all=warning uptime,level,tags
#1: stderr all=off uptime,level,tags
Environment Variables:
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
SHELL=/bin/bash
DISPLAY=localhost:10.0
ARCH=i386
```

#### Note:

The list of environment variables is not the full list but rather a subset of the environment variables that are applicable to the Java VM.

#### **Signal Handlers**

On the Oracle Solaris and Linux operating systems, the next information in the error log is the list of signal handlers, as shown in the following example.

```
Signal Handlers:
sa flags=SA RESTART|SA SIGINFO
```



```
SIGTERM: [libjvm.so+0xb60150],
sa_mask[0]=111111101111111111111111, sa_flags=SA_RESTART|
SA_SIGINFO
SIGQUIT: [libjvm.so+0xb60150],
sa_mask[0]=111111101111111111111111, sa_flags=SA_RESTART|
SA_SIGINFO
```

# System Section Format

The final section in the error log is the system information. The output is operatingsystem-specific but in general includes the operating system version, CPU information, and summary information about the memory configuration.

The following example shows output on a Linux operating system.

----- SYSTEM ------OS:DISTRIB ID=Ubuntu DISTRIB RELEASE=12.04 DISTRIB CODENAME=precise DISTRIB DESCRIPTION="Ubuntu 12.04 LTS" uname:Linux 3.2.0-24-generic #39-Ubuntu SMP Mon May 21 16:52:17 UTC 2012 x86 64 libc:glibc 2.15 NPTL 2.15 rlimit: STACK 8192k, CORE infinity, NPROC 1160369, NOFILE 4096, AS infinity load average:0.46 0.33 0.27 /proc/meminfo: MemTotal: 148545440 kB MemFree: 1020964 kB MemFree: 1020964 kB 

 MemFree:
 1020964 kB

 Buffers:
 29600728 kB

 Cached:
 86607768 kB

 SwapCached:
 16112 kB

 Active:
 52272944 kB

 Inactive:
 64862992 kB

 Active(anon): 314080 kB Inactive(anon): 616296 kB Active(file): 51958864 kB Inactive(file): 64246696 kB Unevictable: 16 kB Mlocked: 16 kB SwapTotal: 1051644 kB Swapfree: Dirty: Writeback: AnonPages: 976092 kB 40 kB 0 kB 912404 kB 95804 kB 2936 kB Mapped: Shmem: 2936 kB Slab: 28625980 kB SReclaimable: 28337400 kB SUnreclaim: 288580 kB KernelStack: 6040 kB PageTables: 42524 kB



NFS_Unstable:	0	kВ	
Bounce:	0	kВ	
WritebackTmp:	0	kВ	
CommitLimit:	75324364	kВ	
Committed AS:	6172612	kВ	
VmallocTotal:	343597383	367	kВ
VmallocUsed:	681668	kВ	
VmallocChunk:	342823793	392	kВ
HardwareCorrupte	ed: 0	kВ	
AnonHugePages:	0	kВ	
HugePages_Total:	0		
HugePages_Free:	0		
HugePages_Rsvd:	0		
HugePages_Surp:	0		
Hugepagesize:	2048	kВ	
DirectMap4k:	171520	kВ	
DirectMap2M:	8208384	kВ	
DirectMap1G:	142606336	6 kE	3

CPU:total 24 (initial active 24) (6 cores per cpu, 2 threads per core) family 6 model 44 stepping 2, cmov, cx8, fxsr, mmx, sse, sse2, sse3, ssse3, sse4.1, sse4.2, popcnt, aes, clmul, ht, tsc, tscinvbit, tscinv CPU Model and flags from /proc/cpuinfo: model name : Intel(R) Xeon(R) CPU X5675 @ 3.07GHz flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpelgb rdtscp lm constant\_tsc arch\_perfmon pebs bts rep\_good nopl xtopology nonstop\_tsc aperfmperf pni pclmulqdq dtes64 monitor ds\_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid dca sse4\_1 sse4\_2 popcnt aes lahf\_lm ida arat epb dts tpr shadow vnmi flexpriority ept vpid

Memory: 4k page, physical 148545440k(1020964k free), swap 1051644k(976092k free)

vm\_info: Java HotSpot(TM) 64-Bit Server VM (9-ea+167) for linux-amd64 JRE (9ea+167), built on Apr 27 2017 00:28:45 by "javare" with gcc 4.9.2

On the Oracle Solaris and Linux, the operating system, information is in the file /etc/ \*release. This file describes the kind of system the application is running on, and in some cases, the information string might include the patch level. Some system upgrades are not reflected in the /etc/\*release file. This is especially true on the Linux system, where the user can rebuild any part of the system.

On Oracle Solaris operating system the uname system call is used to get the name for the kernel. The thread library (T1 or T2) is also printed.

On the Linux system, the uname system call is also used to get the kernel name. The libc version and the thread library type are also printed, as shown in the following example.

uname:Linux 3.2.0-24-generic #39-Ubuntu SMP Mon May 21 16:52:17 UTC 2012 x86\_64 libc:glibc 2.15 NPTL 2.15



On Linux, there are three possible thread types, namely linuxthreads (fixed stack), linuxthreads (floating stack), and NPTL. They are normally installed in / lib, /lib/i686, and /lib/tls.

It is useful to know the thread type. For example, if the crash appears to be related to pthread, then you might be able to work around the issue by selecting a different pthread library. A different pthread library (and libc) can be selected by setting LD\_LIBRARY\_PATH or LD\_ASSUME\_KERNEL.

The glibc version usually does not include the patch level. The command  $\tt rpm$  -q glibc might provide more detailed version information.

On the Oracle Solaris and Linux operating systems, the next information is the rlimit information.

#### Note:

The default stack size of the VM is usually smaller than the system limit, as shown in the following examples:

rlimit: STACK 8192k, CORE infinity, NPROC 1160369, NOFILE 4096, AS
infinity
load average:0.04 0.05 0.02

rlimit Component	Description
STACK 8192k	Stack size (ulimit -s)
CORE infinity	Core dump size (ulimit -c)
NPROC 1160369	Max user processes (ulimit -u)
NOFILE 4096	Max open files (ulimit -n)
AS infinity	Virtual memory (-v)

#### Table A-9 rlimit Description

rlimit: STACK 8192k, CORE 0k, NPROC 4092, NOFILE 1024, AS infinity load average:0.04 0.05 0.02

#### Table A-10 rlimit Description

rlimit Component	Description
STACK 8192k	Stack size (ulimit -s)
CORE 0k	Core dump size (ulimit -c)
NPROC 4092	Max user processes (ulimit -u)
NOFILE 1024	Max open files (ulimit -n)
AS infinity	Virtual memory (-v)



The next information specifies the CPU architecture and capabilities identified by the VM at startup, as shown in the following example.

CPU:total 24 (initial active 24) (6 cores per cpu, 2 threads per core) family 6 model 44 stepping 2, cmov, cx8, fxsr, mmx,sse, sse2, sse3, ssse3, sse4.1, sse4.2, popcnt, aes, clmul, ht, tsc, tscinvbit, tscinv

#### Table A-11 CPU Architecture Description

CPU Architecture Attribute	Description	
CPU:total 24 (initial active 24) (6 cores per cpu, 2 threads per core)	Total number of CPUs	
family 6 model 44 stepping 2	processor family (IA32 only): • 3 - i386	
	<ul> <li>4 - i486</li> <li>5 - Pentium</li> </ul>	
	• 6 - PentiumPro, PII, PIII	
	• 15 - Pentium 4	
cmov, cx8, fxsr, mmx	CPU features	

Table A-12 shows the possible CPU features on a SPARC system.

#### Table A-12 SPARC Features

SPARC Feature	Description
has_v8	Supports v8 instructions.
has_v9	Supports v9 instructions.
has_vis1	Supports visualization instructions.
has_vis2	Supports visualization instructions.
is_ultra3	UltraSparc III.
no-muldiv	No hardware integer multiply and divide.
no-fsmuld	No multiply-add and multiply-subtract instructions.

Table A-13 shows the possible CPU features on an Intel/IA32 system.

#### Table A-13 Intel/IA32 Features

Intel/IA32 Feature	Description
CMOV	Supports cmov instruction.
cx8	Supports cmpxchg8b instruction.
fxsr	Supports fxsave and fxrstor.
mmx	Supports MMX.
sse	Supports SSE extensions.
sse2	Supports SSE2 extensions.
ht	Supports Hyper-Threading Technology.



Table A-14 shows the possible CPU features on an AMD64/EM64T system.

AMD64/EM64T Feature	Description
amd64	AMD Opteron, Athlon64, and so forth.
em64t	Intel EM64T processor.
3dnow	Supports 3DNow extension.
ht	Supports Hyper-Threading Technology.

The next information in the error log is memory information, as shown in the following example.

Memory: 4k page, physical 513604k(11228k free), swap 530104k(497504k free)

#### Table A-15 Memory Configuration Description

Memory Configuration	Description
4k page	Page size
physical 513604k	Total amount of physical memory
(11228k free)	Unused physical memory
swap 530104k	Total amount of swap space
(497504k free)	Unused swap space

Some systems require swap space to be at lease twice the size of real physical memory, whereas other systems do not have any requirements. As a general rule, if both physical memory and swap space are almost full, then there is good reason to suspect that the crash was due to insufficient memory.

On Linux system, the kernel may convert most of unused physical memory to file cache. When there is a need for more memory, the Linux kernel will give the cache memory back to the application. This is handled transparently by the kernel, but it means that the amount of unused physical memory reported by the fatal error handler could be close to zero when there is still sufficient physical memory available.

The final information in the SYSTEM section of the error log is  $vm_info$ , which is a version string embedded in libjvm.so/jvm.dll. Every Java VM has its own unique  $vm_info$  string. If you are in doubt about whether the fatal error log was generated by a particular Java VM, check the version string.



## B Java 2D Properties

This appendix presents properties that can be useful in troubleshooting Java 2D.

This appendix contains the following sections:

- Properties on Oracle Solaris and Linux
- Properties on Windows

### Properties on Oracle Solaris and Linux

List of Java 2D properties on Oracle Solaris and Linux.

Table B-1 describes the default values of some useful properties on Oracle Solaris and Linux platforms.

Setup	DGA	SHM	Pixmap s	OnScreen	OffScreen
Oracle Solaris SPARC with DGA support	On	On	Off	DGA/Software	Software
Oracle Solaris SPARC with no DGA, Oracle Solaris x86, Linux, SunRay, VNC	Off	On	On	X11/MITSHM	Shared/Server Pixmaps
J2SE 1.4 or greater: Remote X server, ssh	Off	Off	On	X11	Server Pixmaps
J2SE 1.3.1 or less: Remote X server, ssh	Off	Off	Off	X11	Software

#### Table B-1 Default Java 2D Properties on Oracle Solaris and Linux

The following list explains how to change the defaults.

- The X11 pipeline is the default pipeline for Oracle Solaris and Linux. Change this default as follows:
  - -Dsun.java2d.opengl=true Attempt to enable the OpenGL pipeline.
- The use of DGA is controlled as follows:
  - NO J2D DGA unset Use DGA, if available.
  - NO J2D DGA set Disable the use of DGA.
- MIT Shared Memory Extension (SHM) is controlled as follows:
  - To use SHM, if available, specify either one of the following properties:

NO\_J2D\_MITSHM unset

J2D\_USE\_MITSHM=true

To not use SHM, specify either one of the following properties:



NO J2D MITSHM set

J2D USE MITSHM=false

- The general use of pixmaps is controlled as follows:
  - -Dsun.java2d.pmoffscreen unset Use pixmaps if DGA is not available.
  - -Dsun.java2d.pmoffscreen=true Force the use of pixmaps.
  - -Dsun.java2d.pmoffscreen=false Disable the use of pixmaps.
- The use of Shared and Server pixmaps is controlled as follows:
  - J2D\_PIXMAPS unset Use both types.
  - J2D PIXMAPS=shared Use only shared memory pixmaps.
  - J2D PIXMAPS=sserver Use only server-side pixmaps.
- The choice of default visual is controlled as follows:
  - FORCEDEFVIS unset (default) Use the best visual available.
  - FORCEDEFVIS set to a hexadecimal value Use the visual whose ID is the hexadecimal value.
  - FORCEDEFVIS set to any other value Use the default visual.

### **Properties on Windows**

The following list describes some useful properties on Windows platforms.

- The DirectDraw/GDI pipeline is the default pipeline for Windows. Change this default as follows:
  - Dsun.java2d.noddraw=true Disable the use of the DirectDraw pipeline.
     GDI will be used instead.
  - - Dsun.java2d.noddraw=false Enable the use of the DirectDraw pipeline.
  - - Dsun.java2d.d3d=false Disable the use of the Direct3D pipeline.
  - J2D D3D=false Disable the use of the Direct3D pipeline.
  - -Dsun.java2d.d3d=true Enable the use of the Direct3D pipeline.
  - J2D D3D=true Enable the use of the Direct3D pipeline.
- Control the use of the built-in surface punting mechanism as follows:
  - - Dsun.java2d.ddforcedram=true Keep volatile images in VRAM.
- Control the use of DirectDraw blit operations as follows:
  - Dsun.java2d.ddblit=false Disable the use of DirectDraw blit operations.
     GDI blits will be used instead.



# C Environment Variables and System Properties

This appendix describes environment variables and system properties that can be useful for troubleshooting problems with the Java HotSpot VM.

Submit a Bug Report contains information on collecting environment variables in Environment Variables.

This appendix contains the following sections:

- The JAVA\_TOOL\_OPTIONS Environment Variable
- The java.security.debug System Property

### The JAVA\_TOOL\_OPTIONS Environment Variable

In many environments, the command line is not readily accessible to start the application with the necessary command-line options.

This often happens with applications that use embedded VMs (meaning they use the Java Native Interface (JNI) Invocation API to start the VM), or where the startup is deeply nested in scripts. In these environments the JAVA\_TOOL\_OPTIONS environment variable can be useful to augment a command line.

#### Note:

In some cases, this option is disabled for security reasons. For example, on the Oracle Solaris operating system, this option is disabled when the effective user or group ID differs from the real ID.

This environment variable allows you to specify the initialization of tools, specifically the launching of native or Java programming language agents using the <code>-agentlib</code> or <code>-javaagent</code> options.

This variable can also be used to augment the command line with other options for diagnostic purposes. For example, you can supply the -XX:OnError option to specify a script or command to be executed when a fatal error occurs.

Because this environment variable is examined at the time, that the JNI\_CreateJavaVM function is called, it cannot be used to augment the command line with options that would normally be handled by the launcher, for example, VM selection using the -client option or the -server option.



### The java.security.debug System Property

This system property controls whether the security system of the Java Runtime Environment (JRE) prints trace messages during execution.

This option can be useful when diagnosing an issue involving a security manager when a SecurityException is thrown.

The java.security.debug property can have the following values:

access

Print all checkPermission results.

The following additional options can be specified with the access option:

stack

Include stack trace.

— domain

Dump all domains in context.

failure

Before throwing an exception, dump the stack and domain that did not have permission.

• jar

Print the JAR verification information.

policy

Print the permissions that SecureClassLoader assigns.

• scl

For example, to print all checkPermission results and trace all domains in context, set the java.security.debug property to access, stack. To trace access failures, set the property to access, failure.

The following example shows the output of a checkPermission failure.

```
$ java -Djava.security.debug="access,failure" MyApp
access denied (java.net.SocketPermission server.foobar.com resolve
)
java.lang.Exception: Stack trace
   at java.lang.Thread.dumpStack(Thread.java:1158)
   at java.security.AccessControlContext.checkPermission
                          (AccessControlContext.java:253)
   at.
java.security.AccessController.checkPermission(AccessController.java
:427)
    at.
java.lang.SecurityManager.checkPermission(SecurityManager.java:532)
   at
java.lang.SecurityManager.checkConnect(SecurityManager.java:1031)
   at java.net.InetAddress.getAllByName0(InetAddress.java:1117)
   at java.net.InetAddress.getAllByName0(InetAddress.java:1098)
```



- at java.net.InetAddress.getAllByName(InetAddress.java:1061)
- at java.net.InetAddress.getByName(InetAddress.java:958)
- at java.net.InetSocketAddress.<init>(InetSocketAddress.java:124)
- at java.net.Socket.<init>(Socket.java:178)
- at MyApp.main(MyApp.java:7)

To know more about the java.security.debug system property, see the Troubleshooting Security in the Java Platform, Standard Edition Security Developer's Guide.

# D Command-Line Options

This appendix describes some command-line options that can be useful when diagnosing problems with the Java HotSpot VM.

This appendix contains the following sections:

- Java HotSpot VM Command-Line Options
- Other Command-Line Options

### Java HotSpot VM Command-Line Options

Command-line options that are prefixed with -xx are specific to the Java HotSpot Virtual Machine. Many of these options are important for performance tuning and diagnostic purposes, and are therefore described in this appendix.

To know more about all possible -xx options, see the Java HotSpot VM Options.

You can dynamically set, unset, or change the value of certain Java VM flags for a specified Java process using the jinfo -flag command. See The jinfo Utility and the JConsole utility.

For a complete list of these flags, use the **MBeans** tab of the JConsole utility. See the list of values for the DiagnosticOptions attribute of the HotSpotDiagnostic MBean, which is in the com.sun.management domain. The following are the flags:

- HeapDumpOnOutOfMemoryError
- HeapDumpPath
- PrintGC
- PrintGCDetails
- PrintGCTimeStamps
- PrintClassHistogram
- PrintConcurrentLocks

#### The -XX:HeapDumpOnOutOfMemoryError Option

This option tells the Java HotSpot VM to generate a heap dump when an allocation from the Java heap or the permanent generation cannot be satisfied. There is no overhead in running with this option, so it can be useful for production systems where the OutOfMemoryError exception takes a long time to appear.

You can also specify this option at runtime with the **MBeans** tab in the **JConsole** utility.

The following example shows the result of running out of memory with this flag set.

```
$ java -XX:+HeapDumpOnOutOfMemoryError -mn256m -mx512m ConsumeHeap
java.lang.OutOfMemoryError: Java heap space
Dumping heap to java pid2262.hprof ...
```



The ConsumeHeap fills the Java heap and runs out of memory. When the java.lang.OutOfMemoryError exception is thrown, a heap dump file is created. In this case the file is 507 MB and is created with the name java\_pid2262.hprof in the current directory.

By default, the heap dump is created in a file called <code>java\_pidpid.hprof</code> in the working directory of the VM, as in the example above. You can specify an alternative file name or directory with the <code>-XX:HeapDumpPath=</code> option. For example <code>-XX:HeapDumpPath=/disk2/dumps</code> will cause the heap dump to be generated in the <code>/disk2/dumps</code> directory.

#### The -XX:OnError Option

When a fatal error occurs, the Java HotSpot VM can optionally execute a usersupplied script or command. The script or command is specified using the – XX:OnError=string command-line option, where string is a single command, or a list of commands separated by semicolons. Within this string, all occurrences of %p are replaced with the current PID, and all occurrences of %% are replaced by a single %. The following examples demonstrate how this option can be used when launching a Java application named MyApp with the java launcher.

• java -XX:OnError="pmap %p" MyApp

On the Oracle Solaris operating system the pmap command displays information about the address space of a process. In the example, if a fatal error occurs, then the pmap command is executed and displays the address space of the current process.

 java -XX:OnError="cat hs\_err\_pid%p.log | mail support@example.com" MyApp

In the example above, the contents of the fatal error log file are mailed to a support alias when a fatal error occurs.

java -XX:OnError="gcore %p; dbx - %p" MyApp

On the Oracle Solaris operating system the gcore command creates a core image of the specified process, and the dbx command launches the debugger. In the example above, the gcore command is executed to create the core image of the current process, and the debugger is started to attach to the process when an unexpected error occurs.

java -XX:OnError="gdb - %p" MyApp

On Linux, the gdb command launches the debugger. In the example above, the gdb debugger is launched and attached to the current process when an unexpected error is encountered.

java -XX:OnError="userdump.exe %p" MyApp

On Windows, the userdump.exe utility creates a crash dump of the specified process. The utility does not ship with Windows and should be downloaded from the Microsoft website as a part of the Microsoft OEM Support Tools package.



In the example, the userdump.exe utility is executed to create a core dump of the current process in case of a fatal error.

#### Note:

The example assumes that the path to the userdump.exe utility is defined in the PATH variable.

To know more about creating crash dumps on Windows, see Collect Crash Dumps on Windows.

#### The -XX:ShowMessageBoxOnError Option

When this option is set and a fatal error occurs, the HotSpot VM will display information about the fatal error and prompt the user to specify whether the native debugger is to be launched. In the case of the Oracle Solaris and Linux operating systems, the output and prompt are sent to the application console (standard input and standard output). In the case of Windows, a Windows message box pops up.

The following example shows a fatal error on a Linux system.

```
=
=
Unexpected Error
-
SIGSEGV (0xb) at pc=0x200000001164db1, pid=10791, tid=1026
Do you want to debug the problem?
To debug, run 'gdb /proc/10791/exe 10791'; then switch to thread 1026
Enter 'yes' to launch gdb automatically (PATH must include gdb)
Otherwise, press RETURN to abort...
```

In this case, a SIGSEGV error occurred, and the user is prompted to specify whether the gdb debugger is to be launched to attach to the process. If the user enters y or yes, thengdb will be launched (assuming it is set in the PATH variable).

On the Oracle Solaris operating system, the message is similar to the Linux example, except that the user is prompted to start the dbx debugger.

On Windows a message box is displayed. If the user clicks **Yes**, the VM will attempt to start the default debugger. This debugger is configured by a registry setting which is described in Collect Crash Dumps on Windows. If Microsoft Visual Studio is installed, the default debugger is typically configured to be msdev.exe.

In the above example, the output includes the PID (pid=10791) and also the thread ID (tid=1026). If the debugger is launched, one of the initial steps in the debugger might be to select the thread and get its stack trace.



When the process is waiting for a response, it is possible to use other tools to get a crash dump or query the state of the process. On the Oracle Solaris operating system, for example, a core dump can be obtained using the gcore utility.

On Windows, a Dr. Watson crash dump can be obtained using the userdump or windbg programs. The windbg utility is included in Microsoft's Debugging Tools for Windows and is described in Collect Crash Dumps on Windows. In windbg, select the **Attach to a Process** menu option, which displays the list of processes and prompts for the PID. The HotSpot VM displays a message box, which includes the PID. After you selected the PID, the .dump /f command can be used to force a crash dump. Figure D-1 is an example crash dump created in a file named crash.dump.

Figure D-1 Example of a Crash Dump Created by windbg

<pre>cdLed: 7e030000 7e0d1000 C \VINCOTS-system32\SUB12.dll cdLed: 7e040000 7e0d1000 C \vinCoTS-system32\VINCM dll cdLed: 6d60000 6d2f8000 c \vinCoTS-system32\VINCM dll cdLed: 6d60000 6d58c000 c \vinCoTS-system32\VINCM dll cdLed: 6d60000 6d58c000 c \vinCoTS-system32\VINCM dll cdLed: 6d60000 70000 6d16000 C \vinCoTS-system32\VINCM dll cdLed: 6d60000 73023000 C \vinCoTS-system32\VINCM dll cdLed: 7300000 5304000 C \vinCoTS-system32\VINCM dll cdLed: 7300000 5304000 C \vinCoTS-system32\VINCM dll cdLed: 6300000 6304400 C \vinCoTS-system32\VINCM dll cdLed: 6300000 6304400 C \vinCoTS-system32\VINCM dll cdLed: 6300000 6304400 C \vinCoTS-system32\VINCM dll cdLed: 7300000 77bc0000 C \vinCoTS-system32\VINCM dll cdLed: 7300000 77bc000 C \vinCoTS-system32\VINCM dll cdLed: 730000 74bc000 C \vinCoTS-system32\VINCM dll cdLed: 73</pre>	emmand - Fid 4012 - Vinobig6.3.0017.0
<pre>cdLed: 7e030000 7e0d1000 C \VINCOTS-system32\SUB12.dll cdLed: 7e040000 7e0d1000 C \vinCoTS-system32\VINCM dll cdLed: 6d60000 6d2f8000 c \vinCoTS-system32\VINCM dll cdLed: 6d60000 6d58c000 c \vinCoTS-system32\VINCM dll cdLed: 6d60000 6d58c000 c \vinCoTS-system32\VINCM dll cdLed: 6d60000 70000 6d16000 C \vinCoTS-system32\VINCM dll cdLed: 6d60000 73023000 C \vinCoTS-system32\VINCM dll cdLed: 7300000 5304000 C \vinCoTS-system32\VINCM dll cdLed: 7300000 5304000 C \vinCoTS-system32\VINCM dll cdLed: 6300000 6304400 C \vinCoTS-system32\VINCM dll cdLed: 6300000 6304400 C \vinCoTS-system32\VINCM dll cdLed: 6300000 6304400 C \vinCoTS-system32\VINCM dll cdLed: 7300000 77bc0000 C \vinCoTS-system32\VINCM dll cdLed: 7300000 77bc000 C \vinCoTS-system32\VINCM dll cdLed: 730000 74bc000 C \vinCoTS-system32\VINCM dll cdLed: 73</pre>	
<pre>colLoad 6d2f0000 6d2f8000 c.vidt.Syrrrbin.bji dl1 colLoad 6d6f000 c.vidt.Syrrrbin.bji dl1 colLoad 6d6f000 c.vidt.Syrrrbin.yov.dl1 colLoad 765f0000 765ac000 c.vVINDOWS-Syrtem37v0122.dl1 colLoad 5d6f0000 c.viNDOWS-Syrtem37v0127401 colLoad 5d6f0000 c.viNDOWS-Syrtem37v01212.dl1 colLoad 5d6f0000 c.viNDOWS-Syrtem37v01212.dl1 colLoad 7f6f0000 7f26f000 c.vVINDOWS-Syrtem37v01212.dl1 colLoad 7f26f000 7f26f000 c.vVINDOWS-syrtem37v051212.dl1 colLoad 7f26f000 7f26f000 c.vVINDOWS-syrtem37v051212.dl1 colLoad 7f26f000 7f26f000 c.vVINDOWS-syrtem37v051212.dl1 colLoad 7f26f000 7f26f000 c.vVINDOWS-syrtem37v051212.dl1 colLoad 7f26f000 7f26f000 c.vVINDOWS-syrtem37v050112.dl1 colLoad 7f26f000 7f26f000 c.vVINDOWS-syrtem37v05012.dl1 colLoad 7f26f000 7f26f000 c.vVINDOWS-syrtem37v05012.dl1 colLoad f126f000 c.vVINDOWS-syrtem37v0502.dl2 c.vINDOWS-syrtem37v0502.dl1 colLoad f126f000 c.vVINDOWS-syrtem37v0502.dl1 colLoad f126f000 c.vVIND</pre>	<pre>[edLoad: 7e090000 7e0d1000 C:\WINDOWS\system32\GDI32.dll</pre>
<pre>cdLoad : 76610000 76616000 C: vipk1:Systeminiverity dl1 cdLoad : 6610000 6636000 c: vipk1:Systeminivery.dl1 cdLoad : 66170000 6636000 c: vipk1:Systeminivery.dl1 cdLoad : 66170000 73023000 C: vipk1:Systeminivery.dl1 cdLoad : 6070000 73023000 C: vipk0:System32:VipSPOL.DWV cdLoad : 77150000 77244000 C: VipK007System32:VipSPOL.DWV cdLoad : 77150000 77244000 C: VipK007System32:Vipk2:dl1 cdLoad : 6070000 63014000 C: VipK007System32:Vipk2:dl1 cdLoad : 5300000 63014000 C: VipK007System32:VipK1:dl1 cdLoad : 77150000 77244000 C: VipK007System32:VipK1:dl1 cdLoad : 77150000 77244000 C: VipK007System32:VipK1:dl1 cdLoad : 77150000 77244000 C: VipK007System32:VipK1:dl1 cdLoad : 77160000 77244000 C: VipK007System32:VipK1:dl1 cdLoad : 77160000 77264000 C: VipK007System32:VipK1:dl1 cdLoad : 7150000 71a5000 C: VipK007System32:VipK1:dl1 cdLoad : 7150000 71a5000 C: VipK007System32:VipK1:dl1 cdLoad : 6430000 6434000 C: VipK007System32:VipK2:dl1 cdLoad : 71a5000 71a5000 C: VipK007System32:VipK2:dl1 cdLoad : 71a5000 C: Vi</pre>	
<pre>bedLad: 6d60000 6d66000 c'\dkl.5\prevbin\exertfy.dll cdLad: 6d70000 6d64000 c'\dkl.5\prevbin\systa.dll cdLad: 7d70000 fid4000 C'\dkl.5\prevbin\systa.dll cdLad: 7d70000 7764000 C'\dkl.5\prevbin\systa.dll cdLad: 7d10000 7764000 C'\dkl.5\prevbin\systa.dll cdLad: 7d10000 7764000 C'\dkl.5\prevbin\systa.dll cdLad: 6d50000 6d51000 C'\dkl.5\prevbin\systa.dll cdLad: 6d550000 6d557000 C'\dkl.5\prevbin\systa.dll cdLad: 7d10000 7764000 C'\dkl.5\prevbin\systa.dll cdLad: 6d550000 6d557000 C'\dkl.5\prevbin\systa.dll cdLad: 7d10000 7764000 C'\dkl.5\prevbin\systa.dll cdLad: 6d550000 6d557000 C'\dkl.5\prevbin\systa.dll cdLad: 6d550000 6d55</pre>	
<pre>bdLod: 6d370000 6d38d000 c:\jdkl.5\jrevbin\jrevbi</pre>	
<pre>bdLadi 6464000 6464000 c \\dots\dots\dots\dots\dots\dots\dots\dot</pre>	
<pre>todLoad : 6d070000 6d16500 C:\jdk1.5\jre\bin\avi.dl1 todLoad : 7635000 763ac000 C:\VINDOWS-System32\NMS12.dl1 todLoad : 7635000 763ac000 C:\VINDOWS-System32\NMS12.dl1 todLoad : 7635000 7724600 C:\VINDOWS-System32\NMS12.dl1 todLoad : 7010000 73bc600 C:\VINDOWS-System32\NCTAN32.dl1 todLoad : 700000 77000 C:\VINDOWS-System32\NCTAN32.dl1 todLoad : 700000 770000 C:\VINDOWS-System32\NCTAN32.dl1 todLoad : 700000 770000 C:\VINDOWS-System32\NCTAN32.dl1 todLoad : 7030000 f0000 f00000 C:\VINDOWS-System32\NcTAN32.dl1 todLoad : 7030000 770000 C:\VINDOWS-System32\NcTAN32.dl1 todLoad : 7030000 7703000 C:\VINDOWS-System32\NcTCAN32.dl1 todLoad : 7030000 f103000 C:\VINDOWS-System32\NCTCAN32.dl1 todLoad</pre>	todLoad: 6d370000_6d38d000c\jdk1_S\jre\bin\java.dll
<pre>ledLoad 7300000 73023000 C:\UNINOFS\System32\VINSPOOL_DRV odLoad 7350000 7340000 C:\UNINOFS\System32\vINSPOL_A11 bdLoad 7350000 5104000 C:\UNINOFS\System32\vINSPAct_A11 bdLoad 51000000 5104000 C:\UNINOFS\System32\vINSPAct_A11 bdLoad 51000000 56000 C:\UNINOFS\System32\VINSPAct_A11 bdLoad 5000000 56000 C:\UNINOFS\System32\VINSPAct_A11 bdLoad 5000000 56000 C:\UNINOFS\System32\VINSPAct_A11 bdLoad 5000000 56000 C:\UNINOFS\System32\VINSPAct_A11 bdLoad 5000000 56000 C:\UNINOFS\System32\VINSPAct_A11 bdLoad 5700000 7207000 C:\UNINOFS\System32\VINSPAct_A11 bdLoad 5700000 70a4000 C:\UNINOFS\System32\VINSPAct_A11 bdLoad 5700000 70a4000 C:\UNINOFS\System32\VINSPAct_A11 bdLoad 570000 70a4000 C:\UNINOFS\System32\VINSPAct_A11 bdLoad 570000 70a4000 C:\UNINOFS\System32\VINSPAct_A11 bdLoad 570000 644000 C:\UNINOFS\System32\VINSPAct_A11 bdLoad 510000 644100 C:\UNINOFS\System32\VINSPAct_A11 bdLoad 510000 6451000 C:\UNINOFS\System32\VINSPAct_A11 bdLoad 510000 6451000 C:\UNINOFS\System32\VINSPAct_A11 bdLoad 510000 71a8000 C:\UNINOFS\System32\VINSPAct_A11 bdLoad 510000 6451000 C:\UNINOFS\System32\VINSPACt_A11 bdLoad 510000001 cc-\UNINOFS\System32\VINSPACt_A11 bdLoad 510000010 cc-\UNINOFS\System32\VINSPACt_A11 bdLoad 51000001</pre>	lodLoad: 6d6a0000 6d6af000 c:\jdk1.5\jre\bin\zip.dl1
<pre>todLoad 76390000 763ac000 C \VINDOWS\System32\NIM32 dll todLoad 7150000 7724000 C \VINDOWS\System32\draw dll todLoad 51000000 51046000 C \VINDOWS\System32\NOTINN32 dll todLoad 5000000 5205000 C \VINDOWS\System32\NOTINN32 dll todLoad 5000000 5205000 C \VINDOWS\System32\NOTINN32 dll todLoad 5000000 5304000 C \VINDOWS\System32\NOTINN32 dll todLoad 5000000 5304000 C \VINDOWS\System32\NOTINN32 dll todLoad 5000000 5304000 C \VINDOWS\System32\NOTINN32 dll todLoad 5000000 540000 C \VINDOWS\System32\NoTINN32 dll todLoad 5000000 54000 C \VINDOWS\System32\NoTINN32 dll todLoad 77340000 7726x000 C \VINDOWS\System32\NoTINN32 dll todLoad 71950000 7124000 C \VINDOWS\System32\NoTINN32 dll todLoad 71950000 7124000 C \VINDOWS\System32\NoTINN32 dll todLoad 510000 6434000 C \VINDOWS\System32\NOTINN32 dll todLoad 510000 6455000 C \VINDOWS\System32\NOTINN32 dll todLoad 51000 6455000 C \VINDOWS\System32\NOTINN32 dll todLoad 51000 6455000 C \VINDOWS\System32\NOTINN32 to 1 = 0010 = set0021 as 000001 = ex-0000002 edx-000000001 = edx-00000005 tip*7175a58 espi0336ffc edp=0336ffl4 iopl=0 = nv up ei pl zr na po nc set0110 togEmakPoint: 1775s58 cc</pre>	odLoad: 6d070800 6d1d6000 C:\jdk1.5\jro\bin\avt.dl1
<pre>lodLoad: 711b0000 77244000 C:\WINDOWS\System32\DCIMAN32 dll odLoad: 71bc0000 73bc6000 C:\WINDOWS\System32\DCIMAN32 dll odLoad: 5c000000 5c0c000 C:\WINDOWS\System32\DCIMAN32 dll odLoad: 5c000000 5c0c000 C:\WINDOWS\System32\DCIMAN32 dll odLoad: 5c00000 7c07000 C:\WINDOWS\System32\DCIMAN32 dll odLoad: 77c0000 7c07000 C:\WINDOWS\System32\DCIMAN32 dll odLoad: 77c0000 7c04000 C:\WINDOWS\System32\DCIMAN32 dll odLoad: 70c0000 7c0700 C:\WINDOWS\System32\DCIMAN32 dll odLoad: 70c0000 7c04000 C:\WINDOWS\System32\DCIMAN32 dll odLoad: 6d20000 6d34000 C:\WINDOWS\System32\DCIMAN32 dll odLoad: 6d30000 6d34000 C:\WINDOWS\System32\DCIMAN32 dll odLoad: 6d50000 6d54000 C:\WINDOWS\System32\DCIMAN32 dll odLoad: 6d50000 6d54000 C:\WINDOWS\System32\DCIMAN32 dll odLoad: 7lab000 7lac5000 C:\WINDOWS\System32\DCIMAN32 dll odLoad: 7lab000 7lac5000 C:\WINDOWS\System32\DCIMAN32 dll odLoad: 7lab000 7lac5000 C:\WINDOWS\System32\DCIMAN32 dll odLoad: 7lab0000 7lac5000 C:\WINDOWS\System32\DCIMAN32 dll odLoad: 7lab0000 7lac5000 C:\WINDOWS\System32\DCIMAN32 dll odLoad: 7lab0000 7lac5000 C:\WINDOWS\System32\DCIMAN32 dll odLoad: 1s50000 6d54000 C:\WINDOWS\System32\DCIMAN32 dll odLoad: 1s0000 7lac5000 C:\WINDOWS\System32\DCIMAN32 dll odLoad: 1s0000 file System32\DCIMAN32 dll odLoad: 1s00000 file System32\DCIMAN32 dll odLoad: 1s000000 file System32\DCIMAN32 dll</pre>	odLoad: 73000000 73023000 C:\VINDOVS\System32\VINSPOOL.DRV
<pre>bedLoad: \$1000000 \$104000 C:\VINDOWS\System32\draw dli bedLoad: 73bc0000 73bc600 C:\VINDOWS\System32\D01N700.Dll bedLoad: 6000000 6c0e8000 C:\VINDOWS\System32\D01N700.Dll bedLoad: 6000000 6c0e8000 C:\VINDOWS\System32\D01N700.Dll bedLoad: 77340000 77bc8000 C:\VINDOWS\system32\D01N700.Dll bedLoad: 77340000 77bc8000 C:\VINDOWS\system32\D01N700.Dll bedLoad: 77340000 77bc8000 C:\VINDOWS\system32\D01N700.Dll bedLoad: 77340000 77bc8000 C:\VINDOWS\system32\D00N1300.Dll bedLoad: 77340000 77bc8000 C:\VINDOWS\system32\D00N1300.Dll bedLoad: 77340000 77bc8000 C:\VINDOWS\System32\D00N1300.Dll bedLoad: 77340000 77bc8000 C:\VINDOWS\System32\D00N1300.Dll bedLoad: 6430000 6d2ed000 C:\VINDOWS\System32\D00N1300.Dll bedLoad: 6430000 6d2ed000 C:\VINDOWS\System32\D00N1300.Dll bedLoad: 6430000 6d2ed000 C:\VINDOWS\System32\D00N1300.Dll bedLoad: 6430000 6d2ed000 C:\VINDOWS\System32\D00N1300.Dll bedLoad: 6430000 71ac5000 C:\VINDOWS\System32\D00N252221D.dll bedLoad: 64550000 6d559000 C:\VINDOWS\System32\D022 def 0000003 eii-000000000 edi-00000000 ser?/idf00 ebx=00000000 edx=00000002 edx=000000000 edi=000000005 ip?7175a58 esp=03361f0c ebp=03361f14 iop1=0 n vu ge ip lz na po nc ser001b ser0023 def023 def0203 def=000000 sei.000000000 edi=00000005 ip?7175a58 esp=03361f0c ebp=03361f14 iop1=0 nv ug ei plz na po nc ser001b ser0023 def023 def0203 def=0000000 sei.000000000 edi=00000005 ip?7175a58 esp=03361f0c ebp=03361f14 iop1=0 nv ug ei plz na po nc ser01b ser023 def023 def023 def0000 sei.000000000000000000000000000000000000</pre>	todLoad: 76350800 763ac000 C:NVINDOVSNSystem32NIMM32.dll
<pre>bedLead: 71bc0000 73bc5000 C:\VINDOWS\System32\DCIMAN32.dll dedLead: 71bc0000 73bc5000 C:\VINDOWS\System32\SynTPFcs.dll dedLead: 6300000 63014000 C:\VINDOWS\System32\VERSION.dll dedLead: 70a70000 70ad4000 C:\VINDOWS\System32\VERSION.dll dedLead: 70a70000 70ad4000 C:\VINDOWS\System32\VERSION.dll dedLead: 70a70000 70ad4000 C:\VINDOWS\System32\VERSION.dll dedLead: 70a70000 77abd000 C:\VINDOWS\System32\VERSION.dll dedLead: 70a70000 77abd000 C:\VINDOWS\System32\VERSION.dll dedLead: 70a70000 77abd000 C:\VINDOWS\System32\VERSION.dll dedLead: 77a70000 77abd000 C:\VINDOWS\System32\VERSION.dll dedLead: 77a70000 77abd000 C:\VINDOWS\System32\VERSION.dll dedLead: 77a70000 77abd000 C:\VINDOWS\System32\VERSION.dll dedLead: 6430000 6d543000 C:\VINDOWS\System32\VES2LLP.dll dedLead: 6430000 6d543000 C:\VINDOWS\System32\VES2LLP.dll dedLead: 71ab000 71ab3000 C:\VINDOWS\System32\VES2LLP.dll dedLead: 71ab000 71ab3000 C:\VINDOWS\System32\VES2LLP.dll dedLead: 71ab000 71ab3000 C:\VINDOWS\System32\VES2LLP.dll dedLead: 71ab000 71ab3000 C:\VINDOWS\System32\VES2LLP.dll dedLead: 71ab0000 71ab3000 C:\VINDOWS\System32\VES2LLP.dll dedLead: 71ab000 010 exc-00000000 edi-00000000 edi-000000006 dedIv000000000 edi-000000000 dedIv0000000000 edi-000000000000000000000000000000000000</pre>	
<pre>bedLoad 5e000000 Schemen C:\VINDOWS\System32\SpTPFes.dll cdLoad 77c00000 C:\VINDOWS\System32\SpTPFes.dll cdLoad 77c00000 72ca000 C:\VINDOWS\System32\SpTPFes.dll cdLoad 77c00000 72ca000 C:\VINDOWS\System32\SpTPFes.dll cdLoad 77c00000 72ca000 C:\VINDOWS\System32\SpTPFes.dll cdLoad 71s00000 71ca1000 C:\VINDOWS\System32\SpTPFes.dll cdLoad 642b0000 6441000 C:\VINDOWS\System32\SpTPFes.dll cdLoad 642b0000 6441000 C:\VINDOWS\System32\SpTPFes.dll cdLoad 642b0000 6441000 C:\VINDOWS\System32\SpTPFes.dll cdLoad 5530000 6453000 C:\VINDOWS\System32\SpTPFes.dll cdLoad 5130000 6453000 C:\VINDOWS\SpTPFes.dll cdLoad 513000 6453000 C:\VINDOWS\SpTPFes.dll cdLoad 513000 6453000 C:\VINDOWS\SpTPFes.dll cdLoad 513000 6453000 C:\VINDOWS\SpTPFes.dll cdLoad 513000 6453000 C:\VINDOWS\SpTPFes.dll cdLoad 51000 6453000 C:\VINDOWS\SpTPFes.dll cdLoad 641000 644100 c:\VINDOWS\SpTPFes.dll cdLoad 6410000000 6414000000 64140000000 colored 64100000000 colored 64100000000 641400000000 6414000000000 colored 64140000000000000 641400000000000000 colored 641000000000 colored</pre>	
<pre>bedLoad : 63000000 63014000 C:\VINDOWS\System32\SymTPEcs dll bdLoad : 7200000 77bc0000 C:\VINDOWS\system32\Shell32.dll bdLoad : 70x70000 77bc0000 C:\VINDOWS\system32\Shell32.dll bdLoad : 70x70000 77bc000 C:\VINDOWS\system32\Shell32.dll bdLoad : 64x0000 C:\VINDOWS\System32\Shell32.dll bdLoad : 71x0000 fdx4000 C:\VINDOWS\System32\Sll22.dll bdLoad : 71x0000 fdx4000 C:\VINDOWS\System32\Sl22.dll bdLoad : 71x0000 fdx5000 C:\VINDOWS\System32\Sl22.dll bdLoad : 71x0000 fdx5000 C:\VINDOWS\System32\Sl22.dll bdLoad : 71x0000 fdx5000 C:\VINDOWS\System32\Sl22.dll bdLoad : 71x0000 fdx6000 C:\VINDOWS\System32\Sl22.dll bdLoad : 61x0000001 cx-0000000 cdx-0000000 edi-00000000 cdi-00000005 ip=77175x58 esp=03361fdc ebp=03361f14 icpl=0 mv up ei pl zn na po nc e1:e00000246 ** EBROR: Symbol file could not be found. Defaulted to export symbols for C:\VINDOWS\System32\std11 dll - td111E0xBreakFoint: ** 100x00001 cx-000201 cx-0000000 cdi-000000000000000000000000000000000000</pre>	
<pre>lodLoad: 77c00000 77c07000 C:\VINDOWS\system37\VERSIOR.dll odLoad: 77a70000 77bc4000 C:\VINDOWS\system37\VERSIOR.dll odLoad: 6420000 6424000 C:\VINDOWS\system37\VERSIOR.dll odLoad: 6420000 6424000 C:\VINDOWS\system37\VERSIOR.dll odLoad: 6430000 6444000 C:\VINDOWS\system37\VERSIOR.dll odLoad: 6430000 6444000 C:\VINDOWS\system37\VERSIOR.dll odLoad: 6430000 6444000 C:\VINDOWS\System37\VERSIOR.dll odLoad: 64530000 655000 C:\VINDOWS\System37\VERSIOR.dll odLoad: 64530000 655000 C:\VINDOWS\System37\VERSIOR.dll odLoad: 64530000 71as8000 C:\VINDOWS\System37\VERSIOR.dll odLoad: 64530000 655000 C:\VINDOWS\System37\VERSIOR.dll odLoad: 71as0000 71as8000 C:\VINDOWS\System32\VERSIOR.dll odLoad: 71as0000 File could not be found. Tefaulted to export symbols for C:\VINDOWS\System32\VERSIOR.dll odload: 1013 dusp /mais the reconvend method of creating a complete memory dump of a user mode</pre>	odLosd: Sc000000 Sc0c000 C:\VINDOVS\System32\D3DIN700.DIL
<pre>lodLoad ?7340000 ?7bca000 C:\VINDOWS\system32\shc132.dll lodLoad ?7ba70000 7ba14000 C:\VINDOWS\system32\shc12adll lodLoad ?71950000 7ba14000 C:\VINDOWS\system32\scccott.Vindows.Common-Controls_6595b64164ccfldf_6.0.2600.1515_x- lodLoad 61220000 64264000 C:\VINDOWS\system32\scccott.Vindows.Common-Controls_6595b64164ccfldf_6.0.2600.1515_x- lodLoad 61220000 6446000 C:\VINDOWS\system32\scccott.Vindows.Common-Controls_6595b64164ccfldf_6.0.2600.1515_x- lodLoad 6120000 6446000 C:\VINDOWS\system32\scccott.Vindows.Common-Controls_6595b64164ccfldf_6.0.2600.1515_x- lodLoad 6120000 6446000 C:\VINDOWS\system32\scccott.Vindows.Common-Controls_6595b64164ccfldf_6.0.2600.1515_x- lodLoad 6120000 64446000 C:\VINDOWS\system32\scccett.Vindows.common-Controls_6595b64164ccfldf_6.0.2600.1515_x- lodLoad 6120000 flotC:\VINDOWS\System32\scccett.Vindows.common-Controls_6595b64164ccfldf_6.0.2600.1515_x- lodLoad 6120000 flotC:\VINDOWS\System32\scccett.Vindows.common-Controls_6595b64164ccfldf_6.0.2600.1515_x- lodLoad 612000000 flotC:\VINDOWS\System32\scccett.Vindows.common-Controls_6595b64164ccfldf_6.0.2600.1515_x- lodLoad 612000000 flotC:\VINDOWS\System32\scccett.Vindows.common-Controls_6595b64164ccfldf_6.0.2600.1515_x- lodLoad 612000000 flotC:\VINDOWS\System32\scccett.Vindows.common-Controls_6595b64164ccfldf_6.0.2600.1515_x- lodLoad 6120000000 flotC:\VINDOWS\System32\scccett.Vindows.common-Controls_6595b64164ccfldf_6.0.2600.1515_x- lodLoad 612000000000000000000000000000000000000</pre>	
<pre>cedLead: 70a70000 70a44000 C:\VINDOWS\system32\SHT4FI dil todLead: 7134000 C:\VINDOWS\system32\compt12.dll cdLead: 77340000 773cb000 C:\VINDOWS\system32\compt12.dll cdLead: 6220000 6d2ed000 C:\VINDOWS\system32\compt12.dll cdLead: 6430000 6d543000 C:\VINDOWS\system32\compt.dll cdLead: 6430000 6d543000 C:\VINDOWS\System32\vintext.dll cdLead: 64530000 6d543000 C:\VINDOWS\System32\Vintext.dll cdLead: 71aa0000 71aa8000 C:\VINDOWS\System32\Vintext.dll cdLead: 71aa0000 71aa8000 C:\VINDOWS\System32\Vintext.dll cdLead: 64530000 6d55000 C:\VINDOWS\System32\Vintext.dll cdLead: 64530000 6d55000 C:\VINDOWS\System32\Vintext.dll cdLead: 64530000 6d55000 C:\VINDOWS\System32\Vintext.dll fac.3ac): Broak instruction exceptice = code 8000003 ei:=00000000 edi=000000005 ip=77175a55 esp=033611cc ebp=0336114 icpl=0</pre>	
<pre>cdLoad: 71950000 71a14000 C:\VIINDOWS\WimSkystem32\corect132.dll cdLoad: 77340000 773cb000 C:\VIINDOWS\wystem32\corect132.dll cdLoad: 6420000 642e4000 C:\jdkl.S\jre\bin\jpeg.dll cdLoad: 6430000 64441000 C:\jdkl.S\jre\bin\jpeg.dll cdLoad: 6430000 64441000 C:\jdkl.S\jre\bin\jpeg.dll cdLoad: 64550000 6455000 C:\VIINDOWS\wystem32\VS2.32.dll cdLoad: 71ab0000 71ac5000 C:\VIINDOWS\Wystem32\VS2.32.dll cdLoad: 64550000 64555000 C:\VIINDOWS\Wystem32\VS2.32.dll cdLoad: 64550000 64555000 C:\VIINDOWS\Wystem32\VS2.32.dll cdLoad: 64550000 64555000 C:\VIINDOWS\Wystem32\VS2.32.dll cdLoad: 64550000 64555000 C:\VIINDOWS\Wystem32\V000003 esi-0000000t edi-00000005 up~71id000 ebx=00000001 ecx=000000002 edx=00000000t edi=00000005 up~71i75a53 esp=03361fcc ebp=03361f14 icpl=0 n vu up ei pl zn na po nc s=001b ess=0023 us=0023 us=0000 eliteN000002 edx=00000005 up~71i75a53 esp=03361fcc ebp=03861f14 icpl=0 n vu up ei pl zn na po nc s=01b ess=0023 us=0023 us=0000 eliteN0000024 ** EBK0R: Synbol file cculd not be found. Defaulted to export synbols for C:\VIIND0W5\System32\utdl1 dll - tdl110bpEreakPoint: 7f75a58 co int 3 e013 dusp /f user.dusp dusp /ma is the reconwend method of creating a ccmplete memory dusp e133 dusp /f user.dusp dusp /ma is the reconwend method of creating a ccmplete memory dusp e133 dusp /ma is the reconwend method of creating a ccmplete memory dusp e133 dusp /ma is the reconwend method of creating a ccmplete memory dusp e133 dusp /ma is the reconwend method of creating a ccmplete memory dusp e133 dusp /ma is the reconwend method of creating a ccmplete memory dusp e133 dusp /ma is the reconwend method of creating a ccmplete memory dusp e133 dusp /ma is the reconwend method of creating a ccmplete memory dusp e133 dusp /ma is the reconwend method of creating a ccmplete memory dusp e133 dusp /ma is the reconwend method of creating a ccmplete memory dusp e134 dusp /ma is the reconvend method of creating a ccmplete memory dusp e134 dusp /ma is the reconvend method of creating a ccmplete memory dusp e134 dusp /ma is dusp /</pre>	
<pre>cdLesd: 77340000 773cb000 C:\TINDOTS\system32\cometl32.dll cdLesd: 6430000 64540000 C:\Tidkl S\gre\bin\jpeg dll cdLesd: 6430000 64540000 C:\Tidkl S\gre\bin\jpeg dll cdLesd: 6430000 64540000 C:\Tidkl S\gre\bin\greytem32\text{vell} ddLesd: 6430000 6454000 C:\Tidkl S\greytem32\text{vell} ddLesd: 6430000 71ac5000 C:\Tidkl S\greytem32\text{vell} ddLesd: 6430000 71ac5000 C:\Tidkl S\greytem32\text{vell} ddLesd: 6430000 71ac5000 C:\Tidkl S\greytem32\text{vell} ddLesd: 6450000 6454000 C:\Tidkl S\greytem32\text{vell} ddLesd: 6450000 64540000 C:\Tidkl S\greytem32\text{vell} ddLesd: 6450000 645400000 C:\Tidkl S\greytem32\text{vell} ddLesd: 6450000 6454000000 C:\Tidkl S\greytem32\text{vell} ddLesd: 6450000 64540000000 C:\Tidkl S\greytem32\text{vell} ddLesd: 645000000 Edi-0000000 Edi-00000000 Edi-0000000 Edi-0000000 Edi-00000000 Edi-0000000 Edi-0000000 Edi-0000000 Edi-0000000 Edi-00000000 Edi-0000000 Edi-000000 Edi-0000000 Edi-</pre>	cdLoad: 70a70000 70ad4000 C:\VINDOVS\system32\SHLVAPI.dll
odLoad 6d200000 6d240000 C.\jdk1 S\jre\bin\jpeg dl1 odLoad 6d30000 6d44000 C.\jdk1 S\jre\bin\jpeg dl1 odLoad 6d30000 6d54000 C.\jdk1 S\jre\bin\jpeg dl1 odLoad 71ab0000 71ac5000 C.\jdk1 S\jre\bin\set dl1 odLoad 5d50000 6d55000 C.\jdk1 S\jre\bin\set dl1 odLoad 6d55000 6d55000 C.\jdk1 S\jre\bin\set dl1 odLoad 6d55000 6d55000 C.\jdk1 S\jre\bin\set dl1 inc 3ac) Froak instruction exception = code 9000003 (first chance) ex-?fidf000 ebx-0000001 ecx-00000002 edx-0000003 esi-0000000t edi-00000005 hp=77175a58 esp=03361foc ebp=03361f14 iopl=0 n v up ei pl zr na po nc ex001b esr-0023 de=0023 is=0000 eff=0000 eff=0000246 ex EEROR Symbol file could not be found. Defaulted to export symbols for C.\WINDOW5\System32\mtd11 dl1 - t01110EgfEreakFoint: 7775a58 co int 3 e013 .duap /f user.duap 	
cdLoad: 64430000 6d444000 C:\fdkl.5\jre\bin\jpsd.dll cdLoad: 64530000 6d543000 C:\fdkl.5\jre\bin\jpsd.dll cdLoad: 71a50000 71ac5000 C:\FJNDOW5\System32\V5232LP.dll cdLoad: 71a50000 71ac5000 C:\FJNDOW5\System32\V523ELP.dll cdLoad: 71a50000 ebx=00000001 ex=0000003 esi-00000000 edi=00000005 ip=7717555 esp=33561cc ebp=03366114 icpl=0 nv up ei pl zr na po nc s=7001b cs=0023 ds=0023 ds=00000 edi=000000246 ** EEROR: Symbol file could not be found. Defaulted to export symbols for C:\FJNDOW5\System32\mtdll.dll - tdl1/EbgBreakFoimt: 177555 co int 3 0133 duap /f user.duap duap /mais the recommend method of creating a complete memory duap duap /mais the recommend method of creating a complete memory duap to a user mode process. *** reating user dump - user full duap up successfully written	
<pre>todLoad: 64530000 6d543000 C:\jdt1.5\jre\bin\set.dl1 bdCoad: 71ab0000 71as8000 C:\vjRL00FS\system32\VS2ELP.dl1 cdLoad: 71ab0000 71as8000 C:\vjRL00FS\system32\VS2ELP.dl1 bdCoad: 64550000 6d559000 C:\vjRL00FS\system32\VS2ELP.dl1 bdCoad: 6450000006 bdF3558 espe0336ffcc ebpe0336ff4 (sol=0 nv up ei pl zrna po nc se01b se*0023 ds*0023 es*0023 is*0038 gs*0000 ef1*00000246 ** ERB00F: Symbol file could not be found. Defaulted to export symbols for C:\VINDOWS\System32\wtdl1 dl1 - td111E0gHeakPoint: 7275585 cc int 3 :0130 duap /f user duap .duap /ma is the recommend method of creating a complete memory duap .duap /ma is the recommend method of creating a complete memory duap .duap rank is the recommend method of creating a complete memory duap .testing user dump - user full duap uup successfully written</pre>	
<pre>todLoad: 71ab0000 71ac5000 C:\UINDOWS\System32\V52_32_d11 codLoad: 71ab0000 71ac5000 C:\UINDOWS\System32\V52EXELP_d11 todLoad: 6d550000 6d559000 C:\UINDOWS\System32\V62EXELP_d11 tac.3ac): Eroak instruction exception = code 80000003 eir-00000004 edi-00000005 upr7175a58 esp+03361foc edp=03361f14 iopl=0 n v up ei pl zr na po nc s=001b ssr0023 ds=0023 ds=0000 efit=000000246 ** EENOR: Symbol file could not be found. Defaulted to export symbols for C:\UINDOWS\System32\utdl1 dll - td1110bpHreakPoint: 7f75a58 cc int 3 :013 .dusp /f user_dusp .dusp /f user_dusp .dusp /f user_dusp .todusp dusp - user full dusp uup successfully written</pre>	odLoad: 64430000 66441000 C:\jdk1.5\jre\bin\jpeg.dli
<pre>lodLoad: 71aa0000 71aa8000 C:\VINDOVS\System32\VS2ETLP.dll odLoad: d555000 (d555000 C:\VINDOVS\System32\VS2ETLP.dll fac.3ac): Erosk instruction exception = code 88000033 (first chance) ex-?fidf000 ebx=0000001 ecx=00000002 edx=000000003 esi=00000004 (edi=00000005 ig=?7175a58 esp=033511cc ebp=0336114 icpl=0</pre>	logLoad: 6d5/30000 6d5/3000 CCNdkLSATENDIN.6d1
<pre>lodLoad: 64550000 6d559000 C:\jdkl.5\jre\bin\mice dl1 lac.3ac): Break instruction exception = code \$0000003 (first chance) ax=7/id(000 ehx=00000000 edx=000000000 edx=000000000 edx=000000000 ip=77175a58 esp=03361foc ebp=03361f14 icpl=0 nv up ei pl zr na po nc s=001b sx=0023 ds=0023 ds=0000 ell=0000246 ** EBNOR: Symbol file could not be found. Defaulted to export symbols for C:\VINDOW5\System32\mtd11 dl1 = id1110bpHreakPoint: 'f75s58 co int 3 'ell3' duap /ma is the recommend method of creating a complete memory duap 'of a user mode process. ''reating user dump - user full dump 'unp successfully written</pre>	
fac.3ac): Eroak instruction exception - code 8000003 (first chance) sw7fifd00 ebx=0000001 ecx=000000003 esi=00000000 edi=00000005 ip=7717555 esp=0336f1cc ebp=0336f1(4 iopl=0 nv up ei pl zr na po nc s=001b ss=0023 ds=0023 es=0023 is=0038 gs=0000 ef1=000000246 ** ENROR: Symbol file could not be found. Defaulted to export symbols for C:\VINDOWS\System32\atdl1 dll - td11/EbgBreakPoint: 775555 cc int 3 :013. :013. dusp /m is the recommend method of creating & complete memory dump . dusp /m is the recommend method of creating & complete memory dump . etain user dump - user full dump up successfully written	
<pre>sax-7id1000 ebx=0000001 ecx=00000002 edx=00000001 edi=00000005 ip=77175558 esp=03361f1cc ebp=03361f14 icpl=0 nv up ei pl zr na po nc s=001b sx=0023 ds=0023 ds=0023 ese=0023 ds=0023 ese=0023 ese=00</pre>	
ip-7717555 esp-03361fcc ebp-03361f4 icol-0 nv up ei pl zr na po nc s001b se:0023 ds=0023 ds=0023 is=0003 gs=0000 efl=00000246 ** ERROR: Symbol file could not be found. Defaulted to export symbols for C:\VINDOWS\System32\mtdll dll - t111[ExpErmeMFPoint: 775555 cc int 3 :013 :013 :013, dusp /f user dump . dusp /ma is the recommend method of creating & complete memory dump of a user mode process. ** reating user dump - user full dump unp successfully written	
<pre>s=001b ss=0023 ds=0023 es=0023 is=0038 gs=0000 efl=00000246 ws EBROR. Symbol file oculd not be found. Defaulted to export symbols for C:\UINDOWS\System32\mtdll dll - tdll/EbgBreakFoint: 717558 cc int 3 :013. :013. dusp /f user duspdusp /ma is the recommend method of creating a ccmplete memory dumpdusp was is the recommend method of creating a ccmplete memory dumpdusp user dump - user full dump ump successfully written</pre>	
tdllTbgBreakPoint: 7775556 co int 3 1013) 1013). duap /f user.duap .duap /ma is the recommend method of creating a complete memory dump of a user mode process. Yeating user.dump - user full dump ump successfully vriten	10-11/2-030 esp-033611CC esp-03361114 1001-0 IN up et pl zr na po nc
<pre>idllIDoBreakFoint: ifISoS6 co int 3 :013) :013) duap /f user duap : duap /ma is the recommend method of creating a complete memory dump : d a user mode process. reating user dump - user full dump ump successfully written</pre>	TODD STOLD (States) Stoles is bord The stole of a second in the stole of the stole
777558°co int 3 :013) :013) .duap /f user duap .duap /mais the recommend method of creating a complete memory duap of a user mode process. reating user dump - user full dump ump successfully written	-* Lance symbol file could not be found. Lefaulted to expert symbols for C. windows systemic atoli of -
<pre>:013) :013) .duap /f user.duap .duap /ma is the recommend method of creating a complete memory duap of a user mode process. reating user.dump - user full dump ump successfully written</pre>	
<pre>(013) dump /f user dump .dump /ma is the recommend method of creating a complete memory dump of a user mode process. reating user dump - user full dump ump successfully written</pre>	
dump /ma is the recommend method of creating a complete memory dump of a user mode process. reating user dump - user full dump ump successfully written	
.duap /ma is the recommend method of creating a complete memory dump of a user mode process. Yeating user dump - user full dump ump successfully written	
of a user mode process. reating user dump - user full dump ump successfully vritten	
reating user dump - user full dump ump successfully written	of a log- and a process
uap successfully vritten	***************************************
uap successfully vritten	reating user duan - user full duan
	uap successfully written

In general, the -XX:+ShowMessageBoxOnError option is more useful in a development environment where the debugger tools are available. The -XX:OnError option is more suitable for production environments where a fixed sequence of commands or scripts are executed when a fatal error occurs.

#### **Other -XX Options**

Several other -xx command-line options can be useful when troubleshooting:

• -XX:OnOutOfMemoryError=string

This option can be used to specify a command or script to execute when an OutOfMemoryError exception is thrown.

• **-XX:ErrorFile**=filename



This option can be used to specify a location for the fatal error log file. See Location of Fatal Error Log.

• -xx:HeapDumpPath=path

This option can be used to specify a location for the heap dump. See The - XX:HeapDumpOnOutOfMemoryError Option.

• **-XX:MaxPermSize**=size

This option can be used to specify the size of the permanent generation memory. See Understand the OutOfMemoryError Exception.

#### -XX:+PrintCommandLineFlags

This option can be used to print all the VM command-line flags. See Collect Data for a Bug Report.

-XX:+PrintConcurrentLocks

This option can be used to cause the Control+Break handler to print a list of concurrent locks owned by each thread.

-XX:+PrintClassHistogram

This option can be used to cause the Control+Break handler to print a heap histogram.

-XX:+PrintGCDetails and-XX:+PrintGCTimeStamps

These options can be used to print detailed information about garbage collection. See The -verbose:gc Option.

-XX:+UseAltSigs

On Oracle Solaris 8 and 9 operating system, this option can be used to instruct the HotSpot VM to use alternate signals to SIGUSR1 and SIGUSR2. See Handle Signals on Oracle Solaris, Linux, and macOS.

-XX:+UseConcMarkSweepGC , -XX:+UseSerialGC and -XX:+UseParallelGC

These options can be used to specify the garbage collection policy to be used. See Working Around Crashes During Garbage Collection.

### **Other Command-Line Options**

In addition to the -xx options, many other command-line options can provide troubleshooting information.

This section describes a few of these options.

#### The -Xcheck:jni Option

This option is useful when diagnosing problems with applications that use the Java Native Interface (JNI). Sometimes, bugs in the native code can cause the HotSpot VM to crash or behave incorrectly.

The -Xcheck:jni option is added to the command line that starts the application, as in the following example:

```
java -Xcheck:jni MyApp
```



The -Xcheck:jni option causes the VM to do additional validation of the use of JNI functions. This includes argument validation and other usage constraints as described below.

#### Note:

The option is not guaranteed to find all invalid arguments or diagnose logic bugs in the application code, but it can help diagnose a large number of such problems.

When a significant usage error is detected, the VM prints a message to the application console or to standard output, prints the stack trace of the offending thread, and stops the VM.

The following example shows a null value was incorrectly passed to a JNI function that does not allow a null value.

The following example shows an incorrect argument that was provided to a JNI function that expects a jfieldID argument.

The following checks are considered indicative of significant problems with the native code:

- The thread doing the call is not attached to the JVM
- The thread doing the call is using the JNIEnv belonging to another thread



- A parameter validation check fails:
  - A jfieldID, or jmethodID, is detected as being invalid. For example:
    - \* Of the wrong type
    - \* Associated with the wrong class
  - A parameter of the wrong type is detected
  - An invalid parameter value is detected. For example:
    - \* NULL where not permitted
    - \* An out-of-bounds array index, or frame capacity
    - \* A non-UTF-8 string
    - \* An invalid JNI reference
    - \* An attempt to use a ReleaseXXX function on a parameter not produced by the corresponding GetXXX function

The following checks only result in warnings being printed:

- A JNI call was made without checking for a pending exception from a previous JNI call, and the current call is not safe when an exception may be pending
- The number of JNI local references existing when a JNI function terminates exceeds the number guaranteed to be available. See the EnsureLocalcapacity function
- A class descriptor is in decorated format (Lname;) when it should not be
- A NULL parameter is allowed, but its use is questionable
- Calling other JNI functions in the scope of Get/ReleasePrimitiveArrayCritical or Get/ ReleaseStringCritical

This non-fatal warning message is shown in the following example.

```
Warning: Calling other JNI functions in the scope of Get/ReleasePrimitiveArrayCritical or Get/ReleaseStringCritical
```

A JNI critical region is created when native code uses the JNI functions

GetPrimitiveArrayCritical or GetStringCritical to obtain a reference to an array or string in the Java heap. The reference is held until the native code calls the corresponding release function. The code between the get and release is called a JNI critical section, and during that time, the HotSpot VM cannot bring the VM to a state that allows garbage collection to occur. The general recommendation is not to use other JNI functions within a JNI critical section, and in particular any JNI function that could potentially cause a deadlock. The warning printed above by the -Xcheck:jni option is thus an indication of a potential issue; it does not always indicate an application bug.

#### The -verbose:class Option

This option enables logging of class loading and unloading.

#### The -verbose:gc Option

This option enables logging of garbage collection (GC) information. It can be combined with other HotSpot VM-specific options such as -XX:+PrintGCDetails and - XX:+PrintGCTimeStamps to get further information about GC. The information output



includes the size of the generations before and after each GC, total size of the heap, the size of objects promoted, and the time taken.

The -verbose:gc option can be dynamically enabled at runtime using the management API or JVM TI. See Custom Diagnostic Tools.

The JConsole monitoring and management tool can also enable or disable the option when the tool is attached to a management VM. See JConsole.

#### The -verbose:jni Option

This option enables the logging of JNI. When a JNI or native method is resolved, the HotSpot VM prints a trace message to the application console (standard output). It also prints a trace message when a native method is registered using the JNI RegisterNative function. The -verbose:jni option can be useful when diagnosing issues with applications that use native libraries.



## E

## Summary of Tools in This Release

This appendix prvoides a summary of tools available in the current release of the JDK, as well as the changes since the previous release.

All the JDK troubleshooting tools that are described in this document are available in JDK 9 on both Oracle Solaris and Linux.

The following JDK troubleshooting tools are also available in JDK 9 on Windows:

- Java Mission Control
- Java Flight Recordings
- How to Produce a Flight Recording
- Inspect a Flight Recording
- jcmd
- JConsole
- Java Virtual Machine
- jdb
- jinfo
- jmap
- jps (not currently available on Windows 98 or Windows ME)
- jrunscript
- jstack
- jstat (not currently available on Windows 98 or Windows ME)
- jstatd (not currently available on Windows 98 or Windows ME)
- visualgc (not currently available on Windows 98 or Windows ME)

