Java Platform, Standard Edition
Troubleshooting Guide

Release 17
F41273-05
April 2023

Java Platform, Standard Edition Troubleshooting Guide, Release 17

F41273-05

Contents

3 Troubleshoot Memory Leaks

4 Troubleshoot Performance Issues Using Flight Recorder

12 Java 2D Pipeline Rendering and Properties

13 Java 2D

E 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.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of

these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Documents

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

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

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.

- Troubleshoot Security APIs

  Provides a link to debug Java security issues.

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

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.

1. **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, core files are

sometimes disabled by default. To enable core files on Linux, it is usually enough to run the `ulimit -c unlimited` 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 enabled.

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, enable this option so that you get this data if the application runs into an unexpected `OutOfMemoryError`. (The option is disabled by default.)

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

   Continuous flight recordings are a circular buffer of Flight Recorder events. If the application runs into an issue, you can dump the data from the last hour of the run. The Flight Recorder 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.

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 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.

5. **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 running it. Another alternative is to add `-XX+PrintCommandLineFlags` and `-showversion` to the JVM arguments.

6. **Set up JMX for remote monitoring:** JMX can be used to connect to a Java application remotely using tools such as JDK 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`.
  - 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 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

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

JDK command-line utilities can be used to diagnose issues and monitor applications that are deployed in the Java runtime environment.

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

JDK Mission Control (JMC) is a production-time profiling and diagnostics tool. It includes tools to monitor and manage your Java application with a very small performance overhead, and is suitable for monitoring applications running in production.

JMC is not part of the regular JDK installation. For more information on JMC downloads and documentation, see JDK Mission Control Page.

JMC consists of :

- *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.
- Plug-ins help in heap dump analysis and DTrace recording. See Plug-in Details. JMC 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 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 as they occur, with a timestamp.
- 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.
4. 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.
5. 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.
7. 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.
4. 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 Development Kit Tool Specifications*.

- 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 command-line diagnostic commands. The simplest way to execute a diagnostic command is to use the `jcmd` tool located in the *`java-home`*`/bin` directory. For more details see, The jcmd 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 Development Kit Tool Specifications*
- 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. These requests are useful for managing recordings from Flight Recorder, troubleshooting, and diagnosing JVM and Java applications.

`jcmd` 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 `jcmd` 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.

Permission: java.lang.management.ManagementPermission(monitor)

Syntax : Thread.print [options]

Options: (options must be specified using the <key> or <key>=<value> syntax)
        -l : [optional] print java.util.concurrent locks (BOOLEAN, false)
        -e : [optional] print extended thread information (BOOLEAN, false)

> jcmd MyProgram Thread.print
2125:
2020-01-21 17:05:10
Full thread dump Java HotSpot(TM) 64-Bit Server VM (14-ea+29-1384 mixed
mode):
...
```

The following sections describe some useful commands and troubleshooting techniques with the `jcmd` utility:

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

Useful Commands for the jcmd Utility

The available diagnostic commands depend on the JVM being used. Use `jcmd <process id/main class> help` to see all available options.

The following are some of the most useful commands of the `jcmd` tool:

- 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` to send diagnostic command requests to a running Java Virtual Machine (JVM) or Java application.

The `jcmd` utility provides the following troubleshooting options:

- Start recording with Flight Recorder.

  For example, to start a 2-minute recording on the running Java process with the identifier `7060` and save it to `C:\TEMP\myrecording.jfr`, 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.

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

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 (reserved=539KB, committed=539KB)
                             (malloc=8KB #109)
                             (tracking overhead=530KB)

-               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=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 sample output for virtual memory for tracking level set to ***detail***, which is shown in addition to the summary output above. One way to get this sample output is to run: `jcmd <pid> VM.native_memory detail`.

```
Virtual memory map:

[0x00000000a1000000 - 0x0000000800000000] reserved 30916608KB for
Java Heap from
    [0x00007f5b91a2472b]
ReservedHeapSpace::try_reserve_heap(unsigned long, unsigned long,
bool, char*)+0x20b
    [0x00007f5b91a24de9]
ReservedHeapSpace::initialize_compressed_heap(unsigned long,
unsigned long, bool)+0x5a9
    [0x00007f5b91a254c6]
ReservedHeapSpace::ReservedHeapSpace(unsigned long, unsigned long,
bool, char const*)+0x176
    [0x00007f5b919da835] Universe::reserve_heap(unsigned long,
unsigned long)+0x65

               [0x00000000a1000000 - 0x0000000117000000] committed
1933312KB from
            [0x00007f5b9132c9be]
G1PageBasedVirtualSpace::commit(unsigned long, unsigned long)+0x18e
            [0x00007f5b913414d1]
G1RegionsLargerThanCommitSizeMapper::commit_regions(unsigned int,
unsigned long, WorkGang*)+0x1a1
            [0x00007f5b913d5c78]
HeapRegionManager::commit_regions(unsigned int, unsigned long,
WorkGang*)+0x58
            [0x00007f5b913d6c45] HeapRegionManager::expand(unsigned
int, unsigned int, WorkGang*)+0x35

               [0x00000007fe000000 - 0x00000007fef00000] committed
15360KB from
            [0x00007f5b9132c9be]
G1PageBasedVirtualSpace::commit(unsigned long, unsigned long)+0x18e
            [0x00007f5b913414d1]
G1RegionsLargerThanCommitSizeMapper::commit_regions(unsigned int,
unsigned long, WorkGang*)+0x1a1
            [0x00007f5b913d5c78]
HeapRegionManager::commit_regions(unsigned int, unsigned long,
WorkGang*)+0x58
            [0x00007f5b913d7355]
HeapRegionManager::expand_exact(unsigned int, unsigned int,
WorkGang*)+0xd5
```

- **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 run`jcmd <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 this shows us changes in memory usage by category:

```
Native Memory Tracking:

Total: reserved=33485260KB +28KB, committed=497784KB +96KB

-                 Java Heap (reserved=30916608KB, committed=393216KB)
                            (mmap: reserved=30916608KB,
committed=393216KB)

-                     Class (reserved=1048702KB, committed=254KB)
                            (classes #507)
                            (  instance classes #421, array classes #86)
                            (malloc=126KB #635)
                            (mmap: reserved=1048576KB, committed=128KB)
                            (  Metadata:   )
                            (    reserved=8192KB, committed=192KB)
                            (    used=118KB)
                            (    free=74KB)
                            (    waste=0KB =0.00%)
                            (  Class space:)
                            (    reserved=1048576KB, committed=128KB)
                            (    used=5KB)
                            (    free=123KB)
                            (    waste=0KB =0.00%)

-                    Thread (reserved=35984KB, committed=1432KB +68KB)
                            (thread #0)
                            (stack: reserved=35896KB, committed=1344KB
+68KB)
                            (malloc=49KB #212)
                            (arena=39KB #68)

-                      Code (reserved=247729KB, committed=7593KB)
                            (malloc=45KB #438)
                            (mmap: reserved=247684KB, committed=7548KB)

-                        GC (reserved=1209971KB, committed=77267KB)
                            (malloc=29183KB #872)
                            (mmap: reserved=1180788KB, committed=48084KB)

-                  Compiler (reserved=168KB, committed=168KB)
                            (malloc=3KB #34)
                            (arena=165KB #5)
```

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 specific memory leaks:

```
[0x00007f5b9175ea8b]
MemBaseline::aggregate_virtual_memory_allocation_sites()+0x11b
[0x00007f5b9175ed68] MemBaseline::baseline_allocation_sites()+0x188
[0x00007f5b9175efff] MemBaseline::baseline(bool)+0x1cf
```

```
[0x00007f5b917d19a4] NMTDCmd::execute(DCmdSource, Thread*)+0x2b4
                                (malloc=1KB type=Native Memory
Tracking +1KB #18 +18)

[0x00007f5b917635b0]
MallocAllocationSiteWalker::do_malloc_site(MallocSite const*)+0x40
[0x00007f5b91740bc8]
MallocSiteTable::walk_malloc_site(MallocSiteWalker*)+0x78
[0x00007f5b9175ec32] MemBaseline::baseline_allocation_sites()+0x52
[0x00007f5b9175efff] MemBaseline::baseline(bool)+0x1cf
                                (malloc=11KB type=Native Memory
Tracking +10KB #156 +136)

[0x00007f5b91a2472b] ReservedHeapSpace::try_reserve_heap(unsigned
long, unsigned long, bool, char*)+0x20b
[0x00007f5b91a24de9]
ReservedHeapSpace::initialize_compressed_heap(unsigned long,
unsigned long, bool)+0x5a9
[0x00007f5b91a254c6] ReservedHeapSpace::ReservedHeapSpace(unsigned
long, unsigned long, bool, char const*)+0x176
[0x00007f5b919da835] Universe::reserve_heap(unsigned long, unsigned
long)+0x65
                                (mmap: reserved=30916608KB,
committed=475136KB +81920KB Type=Java Heap)

[0x00007f5b91804557] thread_native_entry(Thread*)+0xe7
                                (mmap: reserved=34868KB,
committed=1224KB +68KB Type=Thread Stack)

[0x00007f5b91a23c63] ReservedSpace::ReservedSpace(unsigned long,
unsigned long)+0x213
[0x00007f5b912df57c] G1CollectedHeap::create_aux_memory_mapper(char
const*, unsigned long, unsigned long)+0x3c
[0x00007f5b912e4f13] G1CollectedHeap::initialize()+0x333
[0x00007f5b919da5dd] universe_init()+0xbd
                                (mmap: reserved=483072KB,
committed=7424KB +1280KB Type=GC)

[0x00007f5b91a23c63] ReservedSpace::ReservedSpace(unsigned long,
unsigned long)+0x213
[0x00007f5b912df57c] G1CollectedHeap::create_aux_memory_mapper(char
const*, unsigned long, unsigned long)+0x3c
[0x00007f5b912e4e6a] G1CollectedHeap::initialize()+0x28a
[0x00007f5b919da5dd] universe_init()+0xbd
                                (mmap: reserved=60384KB,
committed=928KB +160KB Type=GC)
```

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:

1. Start the JVM with ***summary*** or ***detail*** tracking using the command line option: `-XX:NativeMemoryTracking=summary` or `-XX:NativeMemoryTracking=detail`.

2. 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.

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.

Table 2-1 Native Memory Tracking Memory Categories

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
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

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.

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 `jconsole`*`pid`*, where *`pid`* is the process ID of the application.

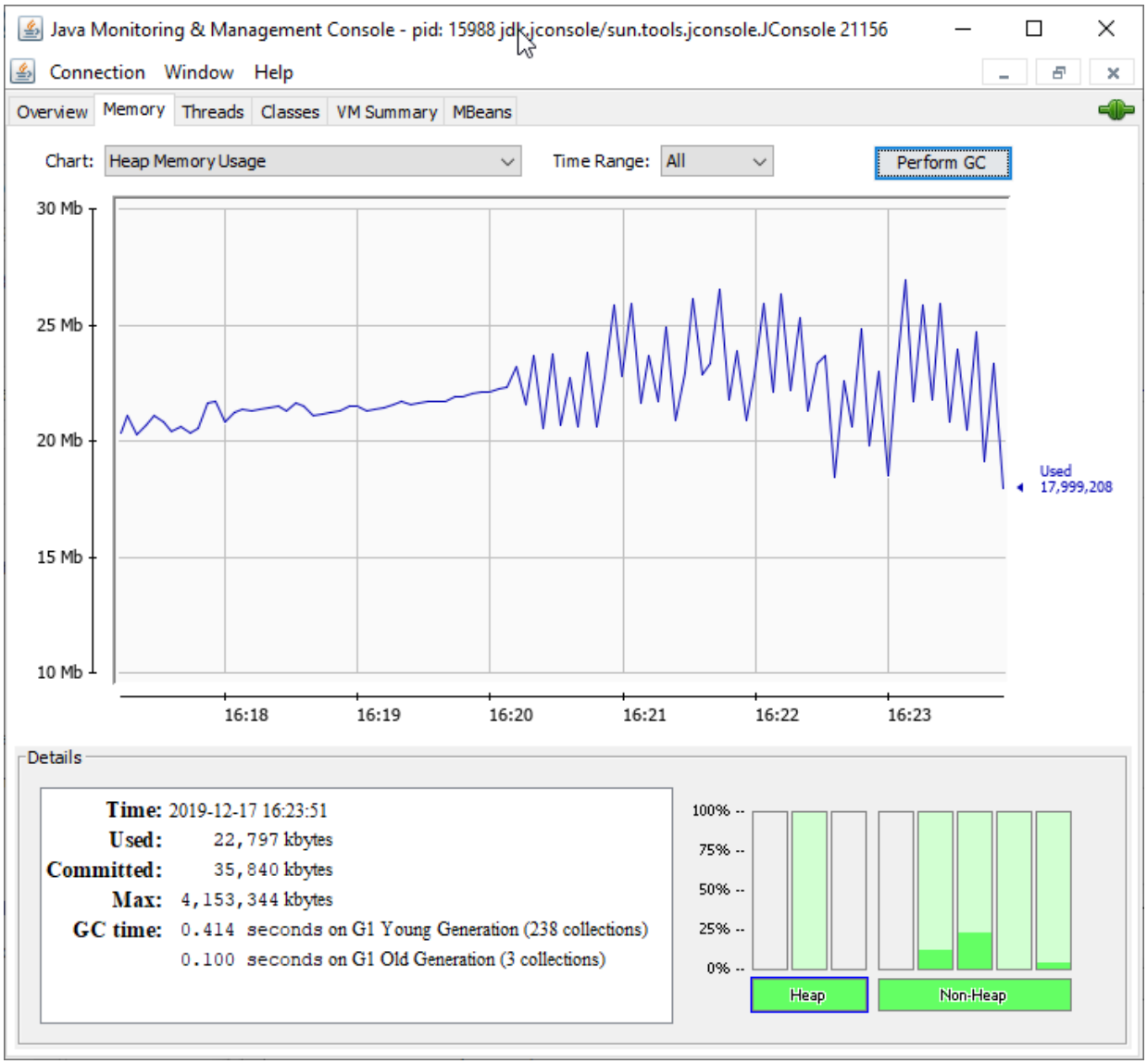
To monitor a remote application, execute the command `jconsole`*`hostname`*`:` *portnumber*, where *`hostname`* is the name of the host running the application, and *`portnumber`* is the port number you specified when you enabled the JMX agent.

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 `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 The jdb Command in the *Java Development Kit Tool Specifications*

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.

JDK Mission Control, Flight Recorder, and the `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 19256 is shown in the following example.

```
c:\Program Files\Java\jdk-13\bin>jinfo 19256
Java System Properties:
java.specification.version=13
sun.cpu.isalist=amd64
sun.jnu.encoding=Cp1252
sun.awt.enableExtraMouseButtons=true
java.class.path=C\:\\sampleApps\\DynamicTreeDemo\\dist\\DynamicTreeDemo.jar
java.vm.vendor=Oracle Corporation
sun.arch.data.model=64
user.variant=
java.vendor.url=https\://java.oracle.com/
os.name=Windows 10
java.vm.specification.version=13
sun.java.launcher=SUN_STANDARD
user.country=US
sun.boot.library.path=C\:\\Program Files\\Java\\jdk-13\\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
java.specification.vendor=Oracle Corporation
java.version.date=2019-09-17
java.home=C\:\\Program Files\\Java\\jdk-13
file.separator=\\
java.vm.compressedOopsMode=Zero based
line.separator=\r\n
java.specification.name=Java Platform API Specification
java.vm.specification.vendor=Oracle Corporation
user.script=
sun.management.compiler=HotSpot 64-Bit Tiered Compilers
java.runtime.version=13-ea+29
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.url.bug=https\://bugreport.java.com/bugreport/
java.io.tmpdir=C\:\\Users\\user1\\AppData\\Local\\Temp\\
java.version=13-ea
user.dir=C\:\\Users\\user1
os.arch=amd64
java.vm.specification.name=Java Virtual Machine Specification
sun.os.patch.level=
java.library.path=C\:\\Program Files\\Java\\jdk-13\\bin;....
java.vm.info=mixed mode, sharing
java.vendor=Oracle Corporation
java.vm.version=13-ea+29
sun.io.unicode.encoding=UnicodeLittle
java.class.version=57.0

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 Linux 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.19256` 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.19256
```

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. 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 memory-related 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.

```
c:\Program Files\Java\jdk-13\bin>jhsdb jmap --heap --pid 19256
Attaching to process ID 19256, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 13-ea+29

using thread-local object allocation.
Garbage-First (G1) GC with 4 thread(s)

Heap Configuration:
   MinHeapFreeRatio         = 40
   MaxHeapFreeRatio         = 70
   MaxHeapSize              = 4253024256 (4056.0MB)
   NewSize                  = 1363144 (1.2999954223632812MB)
   MaxNewSize               = 2551185408 (2433.0MB)
   OldSize                  = 5452592 (5.1999969482421875MB)
   NewRatio                 = 2
   SurvivorRatio            = 8
   MetaspaceSize            = 21807104 (20.796875MB)
   CompressedClassSpaceSize = 1073741824 (1024.0MB)
   MaxMetaspaceSize         = 17592186044415 MB
   G1HeapRegionSize         = 1048576 (1.0MB)

Heap Usage:
G1 Heap:
   regions  = 4056
   capacity = 4253024256 (4056.0MB)
   used     = 7340032 (7.0MB)
   free     = 4245684224 (4049.0MB)
   0.17258382642998027% used
G1 Young Generation:
Eden Space:
   regions  = 7
   capacity = 15728640 (15.0MB)
   used     = 7340032 (7.0MB)
   free     = 8388608 (8.0MB)
   46.666666666666664% used
Survivor Space:
   regions  = 0
   capacity = 0 (0.0MB)
   used     = 0 (0.0MB)
   free     = 0 (0.0MB)
   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 19256.

```
c:\Program Files\Java\jdk-13\bin>jmap -histo 19256
No dump file specified
 num     #instances         #bytes  class name (module)
-------------------------------------------------------
   1:         20913        1658720  [B (java.base@13-ea)
   2:          3647        1516888  [I (java.base@13-ea)
   3:         12321         492840  java.security.AccessControlContext
(java.base@13-ea)
   4:         14806         355344  java.lang.String (java.base@13-ea)
   5:          2441         298464  java.lang.Class (java.base@13-ea)
   6:          5169         289464
jdk.internal.org.objectweb.asm.SymbolTable$Entry (java.base@13-ea)
   7:          5896         284216  [Ljava.lang.Object; (java.base@13-ea)
   8:          6887         220384  java.util.HashMap$Node (java.base@13-ea)
   9:           237         194640
[Ljdk.internal.org.objectweb.asm.SymbolTable$Entry; (java.base@13-ea)
  10:          5119         163808  java.util.ArrayList$Itr (java.base@13-ea)
  11:          1922         153760  java.awt.event.MouseEvent
(java.desktop@13-ea)
  12:           672         139776  sun.java2d.SunGraphics2D
(java.desktop@13-ea)
  13:          4101         131232  java.lang.ref.WeakReference
(java.base@13-ea)
  14:           655         101848  [Ljava.util.HashMap$Node; (java.base@13-
ea)
  15:          3915          93960  sun.awt.EventQueueItem (java.desktop@13-
ea)
  16:           367          89008  [C (java.base@13-ea)
  17:          3708          88992  java.awt.Point (java.desktop@13-ea)
  18:          2158          86320  java.lang.invoke.MethodType
(java.base@13-ea)
  19:          3026          81832  [Ljava.lang.Class; (java.base@13-ea)
  20:           348          77952
jdk.internal.org.objectweb.asm.MethodWriter (java.base@13-ea)
  21:          1016          73152  java.awt.geom.AffineTransform
(java.desktop@13-ea)
  22:          1017          65088  java.awt.event.InvocationEvent
```

```
(java.desktop@13-ea)
  23:          2013          64416  java.awt.Rectangle
(java.desktop@13-ea)
  24:          1341          64368  java.lang.invoke.MemberName
(java.base@13-ea)
  25:          1849          59168
java.util.concurrent.ConcurrentHashMap$Node (java.base@13-ea)
... more lines removed here to reduce output...
1414:             1             16
sun.util.resources.LocaleData$LocaleDataStrategy (java.base@13-ea)
1415:             1             16
sun.util.resources.provider.NonBaseLocaleDataMetaInfo
(jdk.localedata@13-ea)
Total        145508        8388608
```

When the `jhsdb jmap --histo` command is executed on a core file, the tool prints the 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 `jhsdb jmap --histo` command when it is executed on a core file.

```
& jhsdb jmap --exe /usr/java/jdk_12/bin/java --core core.16395 --histo
Attaching to core core.16395 from executable /usr/java/jdk_12/bin/java
please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 12-ea+30
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.util.concurrent.ConcurrentHashMap$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 `jmap` command with the `-clstats` 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
- **ROAll** - Size of class metadata that could be put in read-only memory
- **RWAll** - 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 14400.

```
c:\Program Files\Java\jdk-13\bin>jmap -clstats 11848
Index Super InstBytes KlassBytes annotations   CpAll MethodCount Bytecodes
MethodAll   ROAll    RWAll    Total ClassName
    1    -1    313192        512           0       0           0
0         0       24      624      648 [B
    2    51    287648        784           0   23344         147
5815     52456   28960    50248    79208 java.lang.Class
    3    -1    259936        512           0       0           0
0         0       24      624      648 [I
    4    51    171000        680         136   16304         120
4831     48024   22408    44680    67088 java.lang.String
    5    -1    147200        512           0       0           0
0         0       24      624      648 [Ljava.lang.Object;
    6    51    123680        600           0    1384           7
149      1888    1200     3024     4224 java.util.HashMap$Node
    7    51     53440        608           0    1360           9
213      2472    1632     3184     4816
java.util.concurrent.ConcurrentHashMap$Node
    8    -1     51832        512           0       0           0
0         0       24      624      648 [C
    9    -1     49904        512           0       0           0
0         0       32      624      656 [Ljava.util.HashMap$Node;
```

```
   10     51      31200         624           0    1512           8
240       2224     1472      3256      4728 java.util.Hashtable$Entry
   11     51      25536         592           0   11520          89
4365      48344    16696     45480     62176 java.lang.invoke.MemberName
   12   1614      19296        1024           0    7904          51
4071      30304    14664     25760     40424 java.util.HashMap
   13     -1      18368         512           0       0
0          0          0       32       624       656
[Ljava.util.concurrent.ConcurrentHashMap$Node;
   14     51      17504         544         120    5464          37
1783      14968     7416     14392     21808
java.lang.invoke.LambdaForm$Name
   15     -1      16680         512           0       0
0          0          0       80       624       704 [Ljava.lang.Class;
...lines removed to reduce output... 2342  1972         0
560            0    1912           7       170      1520    1312
3016      4328 sun.util.logging.internal.LoggingProviderImpl
 2343     51          0         528           0     232
1          0       144      128       936     1064
sun.util.logging.internal.LoggingProviderImpl$LogManagerAccess
               2081120    1635072       10680 5108776       27932
1288637   7813992 5420704 10014136 15434840 Total
                 13.5%      10.6%        0.1%   33.1%           -
8.3%      50.6%   35.1%     64.9%   100.0%
Index Super InstBytes KlassBytes annotations   CpAll MethodCount
Bytecodes MethodAll   ROAll    RWAll    Total ClassName
```

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.

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 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 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

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 `-l` 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 14-ea+29-1384
Deadlock Detection:

No deadlocks found.

----------------- 0 -----------------
----------------- 1 -----------------
"DestroyJavaVM" #18 prio=5 tid=0x000001df4706f000 nid=0x744 waiting on
condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
   JavaThread state: _thread_blocked
0x00007ffa4529f9f4      ntdll!ZwWaitForAlertByThreadId + 0x14
0x000001df2533dc50              ????????
----------------- 2 -----------------
0x00007ffa4529c144      ntdll!NtWaitForSingleObject + 0x14
----------------- 3 -----------------
0x00007ffa4529f9f4      ntdll!ZwWaitForAlertByThreadId + 0x14
----------------- 4 -----------------
0x00007ffa4529c144      ntdll!NtWaitForSingleObject + 0x14
----------------- 5 -----------------
0x00007ffa4529f9f4      ntdll!ZwWaitForAlertByThreadId + 0x14
----------------- 6 -----------------
0x00007ffa4529f9f4      ntdll!ZwWaitForAlertByThreadId + 0x14
----------------- 7 -----------------
0x00007ffa4529f9f4      ntdll!ZwWaitForAlertByThreadId + 0x14
----------------- 8 -----------------
"Reference Handler" #2 daemon prio=10 tid=0x000001df47020000
nid=0x4728 waiting on condition [0x000000a733aff000]
   java.lang.Thread.State: RUNNABLE
   JavaThread state: _thread_blocked
0x00007ffa4529f9f4      ntdll!ZwWaitForAlertByThreadId + 0x14
0x000001df2533e280              ????????
----------------- 9 -----------------
"Finalizer" #3 daemon prio=8 tid=0x000001df4702b000 nid=0x5278 in
Object.wait() [0x000000a733bfe000]
   java.lang.Thread.State: WAITING (on object monitor)
   JavaThread state: _thread_blocked
0x00007ffa4529c144      ntdll!NtWaitForSingleObject + 0x14
----------------- 10 -----------------
"Signal Dispatcher" #4 daemon prio=9 tid=0x000001df47053800 nid=0xac0
runnable [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
```

```
   JavaThread state: _thread_blocked
0x00007ffa4529c144      ntdll!NtWaitForSingleObject + 0x14
----------------- 11 -----------------
"Attach Listener" #5 daemon prio=5 tid=0x000001df47058800 nid=0x3980
runnable [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
   JavaThread state: _thread_blocked
0x00007ffa4529c144      ntdll!NtWaitForSingleObject + 0x14
0x000001df47059390              ????????
----------------- 12 -----------------
"Service Thread" #6 daemon prio=9 tid=0x000001df4705b800 nid=0x3350 runnable
[0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
   JavaThread state: _thread_blocked
0x00007ffa4529f9f4      ntdll!ZwWaitForAlertByThreadId + 0x14
----------------- 13 -----------------
"C2 CompilerThread0" #7 daemon prio=9 tid=0x000001df47068800 nid=0x51e8
waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
   JavaThread state: _thread_blocked
0x00007ffa4529f9f4      ntdll!ZwWaitForAlertByThreadId + 0x14
0x000001df2533d590              ????????
----------------- 14 -----------------
"C1 CompilerThread0" #9 daemon prio=9 tid=0x000001df4705d800 nid=0xc20
waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
   JavaThread state: _thread_blocked
0x00007ffa4529f9f4      ntdll!ZwWaitForAlertByThreadId + 0x14
0x000001df2533d590              ????????
----------------- 15 -----------------
"Sweeper thread" #10 daemon prio=9 tid=0x000001df4706c000 nid=0x1a64
runnable [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
   JavaThread state: _thread_blocked
0x00007ffa4529f9f4      ntdll!ZwWaitForAlertByThreadId + 0x14
----------------- 16 -----------------
"Notification Thread" #11 daemon prio=9 tid=0x000001df47070000 nid=0xddc
runnable [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
   JavaThread state: _thread_blocked
0x00007ffa4529f9f4      ntdll!ZwWaitForAlertByThreadId + 0x14
----------------- 17 -----------------
0x00007ffa4529f9f4      ntdll!ZwWaitForAlertByThreadId + 0x14
0x00000f3e40772a94              ????????
----------------- 18 -----------------
"Common-Cleaner" #12 daemon prio=8 tid=0x000001df4706b000 nid=0x2054 in
Object.wait() [0x000000a7344fe000]
   java.lang.Thread.State: TIMED_WAITING (on object monitor)
   JavaThread state: _thread_blocked
0x00007ffa4529c144      ntdll!NtWaitForSingleObject + 0x14
----------------- 19 -----------------
"Java2D Disposer" #13 daemon prio=10 tid=0x000001df4706c800 nid=0x4770 in
Object.wait() [0x000000a7345ff000]
   java.lang.Thread.State: WAITING (on object monitor)
   JavaThread state: _thread_blocked
```

```
0x00007ffa4529c144      ntdll!NtWaitForSingleObject + 0x14
----------------- 20 -----------------
"AWT-Shutdown" #14 prio=5 tid=0x000001df4706d800 nid=0x4ed4 in
Object.wait() [0x000000a7346fe000]
   java.lang.Thread.State: WAITING (on object monitor)
   JavaThread state: _thread_blocked
0x00007ffa4529c144      ntdll!NtWaitForSingleObject + 0x14
----------------- 21 -----------------
"AWT-Windows" #15 daemon prio=6 tid=0x000001df4706e800 nid=0x15e8
runnable [0x000000a7347ff000]
   java.lang.Thread.State: RUNNABLE
   JavaThread state: _thread_in_native
----------------- 22 -----------------
"AWT-EventQueue-0" #17 prio=6 tid=0x000001df4706a000 nid=0x2f54
waiting on condition [0x000000a7348fe000]
   java.lang.Thread.State: WAITING (parking)
   JavaThread state: _thread_blocked
0x00007ffa4529c144      ntdll!NtWaitForSingleObject + 0x14
----------------- 23 -----------------
----------------- 24 -----------------
----------------- 25 -----------------
```

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 `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 The jstat Command in the *Java Development Kit Tool Specifications*.

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 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 `-gcutil` option, where the `jstat` utility attaches to LVMID number 2834 and takes 7 samples at 250-millisecond intervals.

```
$ jstat -gcutil 2834 250 7
  S0     S1     E      O      M     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 `-gcnew` option where the `jstat` utility attaches to LVMID number 2834, takes samples at 250-millisecond intervals, and displays the output. In addition, it uses the `-h3` option to display the column headers after every 3 lines of data.

```
$ jstat -gcnew -h3 2834 250
S0C    S1C    S0U    S1U   TT 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
 S0C    S1C    S0U    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
 S0C    S1C    S0U    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 `-gcoldcapacity` option, where the `jstat` utility attaches to LVMID number 21891 and takes 3 samples at 250-millisecond intervals. The `-t` 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 `-gcoldcapacity` 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 81st 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

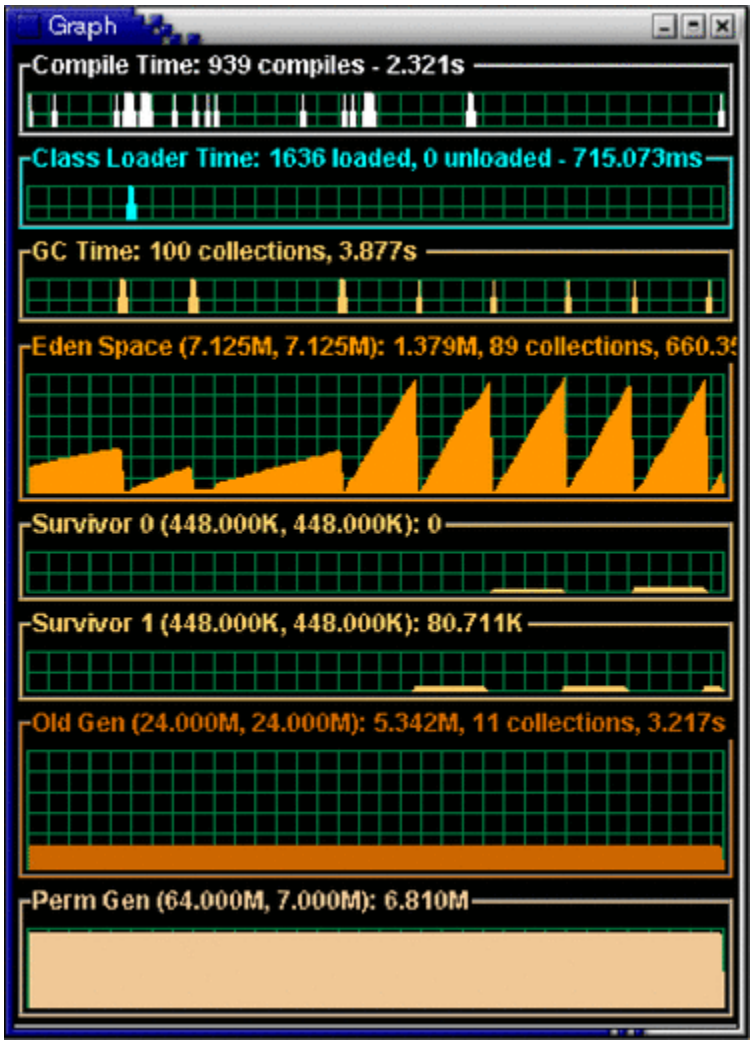
The `visualgc` tool provides a graphical view of the garbage collection (GC) system.

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

On 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 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
[0x00000000..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
[0x00000000..0x00000000]
   java.lang.Thread.State: RUNNABLE

"CompilerThread0" daemon prio=10 tid=0x000c5400 nid=0x7 waiting on
condition [0x00000000..0x00000000]
   java.lang.Thread.State: RUNNABLE

"Signal Dispatcher" daemon prio=10 tid=0x000c4400 nid=0x6 waiting on
condition [0x00000000..0x00000000]
   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`.

```
Heap
 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
```

```
00)
   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

Windows and Linux operating systems provide native tools 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 Linux operating system.

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
- Probe Providers in Java HotSpot VM

Troubleshooting Tools Based on the Operating System

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

Table 2-3 lists the troubleshooting tools available on the Windows operating system.

Table 2-3 Native Troubleshooting Tools on Windows

Tool	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` debugger	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.

Table 2-3 (Cont.) Native Troubleshooting Tools on Windows

Tool	Description
`/Md` and `/Mdd` compiler options	Compiler options that automatically include extra support for tracking memory allocations

Table 2-4 describes some troubleshooting tools introduced or improved in the Linux operating system version 10.

Table 2-4 Native Troubleshooting Tools on Linux

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 Not all distributions provide this tool by default; therefore, you might have to download it from SourceForge.
`ltrace`	Library call tracer Not all distributions provide this tool by default; therefore, you might have to download it from SourceForge.
`mtrace` and `muntrace`	GNU `malloc` tracer
`/proc` filesystem	Virtual filesystem that contains information about processes and other system information
`strace`	System call tracer
`top`	Display most CPU-intensive processes.
`vmstat`	Report information about processes, memory, paging, block I/O, traps, and CPU activity.

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 `hotspot_jni` 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.

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 Java runtime environment.

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

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.

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 2-5 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.

Table 2-5 Postmortem Diagnostics Tools

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:+HeapDumpOnOutOfMemoryError` option	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:+ShowMessageBoxOnError` option	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.

Table 2-5 (Cont.) Postmortem Diagnostics Tools

Tool or Option	Description and Usage
`jhsdb jinfo` utility	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` utility	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.
`jstack` utility	This utility can get Java and native stack information from a Java process. On the Linux operating system, 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.

Hung Processes Tools

Tools and options for diagnosing problems between the application and the Java HotSpot VM in a hung process are available in the JDK and in the operating system.

Table 2-6 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.

JDK Mission Control, 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 2-6 Hung ProcessTools

Tool or Option	Description and Usage
Ctrl+Break handler (Control+\ or `kill -QUIT` *pid* on the and Linux operating system, 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 recordings from Flight Recorder. The recordings 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` utility	This utility can get configuration information from a Java process. See The jinfo Utility.

Table 2-6 (Cont.) Hung ProcessTools

Tool or Option	Description and Usage
`jmap` utility	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.

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 2-7 are designed for monitoring applications that are running.

JDK Mission Control, 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 2-7 Monitoring Tools

Tool or Option	Description and Usage
JDK Mission Control	JDK Mission Control (JMC) is a 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.
`jcmd utility`	This utility is used to send diagnostic command requests to the JVM, where these requests are useful for controlling recordings from Flight Recorder. The recordings 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` utility	This utility can get memory map information, including a heap histogram, from a Java process or a core file. See The jmap Utility.
`jps` utility	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` utility	This utility can get Java and native stack information from a Java process or a core file. See The jstack Utility.

Table 2-7 (Cont.) Monitoring Tools

Tool or Option	Description and Usage
`jstat` utility	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` daemon	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` utility	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. See Native Operating System Tools.

Other Tools, Options, Variables, and Properties

General troubleshooting tools, options, variables, and properties that can help to diagnose issues are available in the JDK and in the operating system.

In addition to the tools that are designed for specific types of problems, the tools, options, variables, and properties listed in Table 2-8 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.

Table 2-8 General Troubleshooting Tools and Options

Tool or Option	Description and Usage
JDK Mission Control	JDK Mission Control (JMC) is a 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.
`jcmd` utility	This utility is used to send diagnostic command requests to the JVM, where these requests are useful for controlling recordings from Flight Recorder. The recordings are used to troubleshoot and diagnose JVM and Java applications with flight recording events.
`jinfo` utility	This utility can dynamically set, unset, and change the values of certain JVM flags for a specified Java process. On Linux operating systems, it can also print configuration information.
`jrunscript` utility	This utility is a command-line script shell, which supports both interactive and batch-mode script execution.

Table 2-8 (Cont.) General Troubleshooting Tools and Options

Tool or Option	Description and Usage
`-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.
`-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 Java runtime environment print trace messages during execution. See The java.security.debug System Property.

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.

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:

The java.lang.OutOfMemoryError Error

One common indication of a memory leak is the `java.lang.OutOfMemoryError` error. This error indicates that the garbage collector cannot make space available to accommodate a new object, and the heap cannot be expanded further. This error may also be thrown when there is insufficient native memory to support the loading of a Java class. In rare instances, the error is thrown when an excessive amount of time is being spent performing garbage collection, and little memory is being freed.

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

A stack trace is printed when a `java.lang.OutOfMemoryError` error is thrown.

An early step to diagnose a `java.lang.OutOfMemoryError` error is to determine its cause. Was it thrown because the Java heap is full or because the native heap is full? To help you discover the cause, a *detail message* is appended to the text of the exception, as shown in the following examples:

Detail Message: Java heap space
Cause: The detail 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. The initial and maximum size of the Java heap space can be configured using the `-Xms` and `-Xmx` options.
In other cases, and in particular for a long-lived application, the message might indicate that the application is unintentionally holding references to objects, which 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 unintentionally be 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 implementations of the Java Runtime, finalizers are executed by a daemon thread that services the finalization queue. If the thread cannot keep up with the finalization queue, then the Java heap could fill up, and this kind of `java.lang.OutOfMemoryError` error 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 Monitoring 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.

Detail Message: GC Overhead limit exceeded
Cause: The detail message **GC overhead limit exceeded** indicates that the garbage collector (GC) is running most of the time, and the Java application is making very slow progress. After a garbage collection, if the Java application spends more than approximately 98% of its time performing garbage collection and if it is recovering less than 2% of the heap and has been doing so for the last five (compile-time constant) consecutive garbage collections, then a `java.lang.OutOfMemoryError` error is thrown. This error is typically thrown because the amount of live data barely fits into the Java heap leaving little free space for new allocations.

Action: Increase the heap size. The `java.lang.OutOfMemoryError` error for **GC Overhead limit exceeded** can be turned off using the command-line flag `-XX:-UseGCOverheadLimit`.

Detail Message: 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

Detail Message: Metaspace
Cause: Java class metadata (the virtual machine's internal presentation of a Java class) is allocated in native memory (referred to here as Metaspace). If the Metaspace for class metadata is exhausted, a `java.lang.OutOfMemoryError` error with a detail message **Metaspace** is thrown. The amount of Metaspace that can be used for class metadata is limited by the parameter `MaxMetaSpaceSize`, which can be specified on the command line. When the amount of native memory needed for class metadata

exceeds `MaxMetaSpaceSize`, a `java.lang.OutOfMemoryError` error with detail message **Metaspace** is thrown.

Action: If `MaxMetaSpaceSize` has been specified on the command-line, increase its value. Metaspace is allocated from the same address space as the Java heap. Reducing the size of the Java heap will make more space available for Metaspace. This trade-off is only useful if there is an excess of free space in the Java heap. See the following action for the **Out of swap space** detail message.

Detail Message: 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 a `java.lang.OutOfMemoryError` error. However, Java reports this apparent error 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 a source module reporting the allocation failure, although sometimes it indicates the actual reason.

Action: When this error is thrown, the Java VM (JVM) invokes the fatal error handling mechanism: 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.
You might need to use troubleshooting utilities for the operating system to diagnose the issue further. See Native Operating System Tools.

Detail Message: 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` (`true` by default). If `UseCompressedClassPointers` is `true`, 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` error with detail message **Compressed class space** is thrown.

Action: Increase `CompressedClassSpaceSize` or set `UseCompressedClassPointers` to `false`.

> **Note:**
>
> There are bounds on the acceptable size of `CompressedClassSpaceSize`. For example `-XX:CompressedClassSpaceSize=4g`, exceeds acceptable bounds and 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`. Other metadata is stored in Metaspace.

Detail Message: *reason stack_trace* (Native method)
Cause: This detail message indicates 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 itself.

Action: If this type of `java.lang.OutOfMemoryError` error is thrown, you might need to use native utilities of the operating system to diagnose the issue further. See Native Operating System Tools.

Detecting a Memory Leak

The `java.lang.OutOfMemoryError` error can be an indication of a memory leak in a Java application. It might also indicate that either the Java heap, Metaspace or the Compressed Class space is sized smaller than the memory requirements of the application for that specific memory pool. Before assuming a memory leak in the application, ensure that the memory pool for which you are seeing the `java.lang.OutOfMemoryError` error is sized adequately. The Java heap can be sized using the `-Xmx` and `-Xms` command-line options, and the maximum and initial size of Metaspace can be configured using `MaxMetaspaceSize` and `MetaspaceSize`. Similarly, the Compressed Class space can be sized using the `CompressedClassSpaceSize` option.

Memory leaks are often very difficult to detect, especially those that are slow. A memory leak occurs when an application unintentionally holds references to Java objects or classes, preventing them from being garbage collected. These unintentionally held objects or classes can grow in memory over time, eventually filling up the entire Java heap or Metaspace, causing frequent garbage collections and eventual process termination with a `java.lang.OutOfMemoryError` error.

For detecting memory leaks, it is important to monitor the live set of the application that is, the amount of Java heap space or Metaspace being used after a full garbage collection. If the live set increases over time after the application has reached a stable state and is under a stable load, that could be a strong indication of a memory leak. The live set and memory usage of an application can be monitored by using **JConsole** and **JDK Mission Control**. The memory usage information can also be extracted from garbage collection logs.

Note that if the detail message of the error suggests the exhaustion of the native heap, the application could be encountering a native memory leak. To confirm native memory leaks, use native tools such as **pmap** or **PerfMon**, and compare their periodically-collected output to determine the newly-allocated or growing memory sections of the process.

JConsole

JConsole is a great tool for monitoring resources of Java applications. Among other things, it is helpful in monitoring the usage of various memory pools of an application, including generations of Java heap, Metaspace, Compressed Class Space, and CodeHeap.

In the following screenshots, for an example program, the JConsole shows the usage of Heap Memory and Old Generation steadily increasing over a period of time. This steady growth in memory usage even after several full garbage collections indicates a memory leak.

Figure 3-1 JConsole Heap Memory

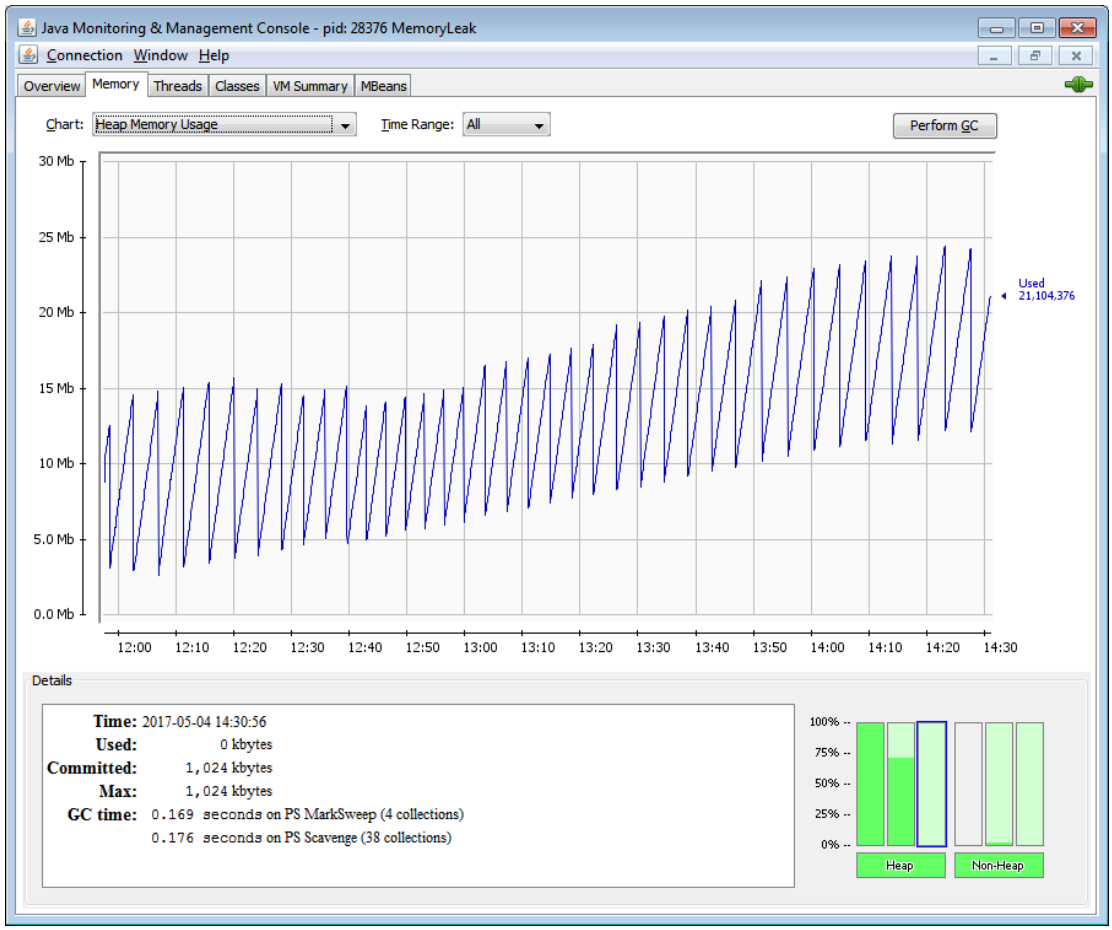
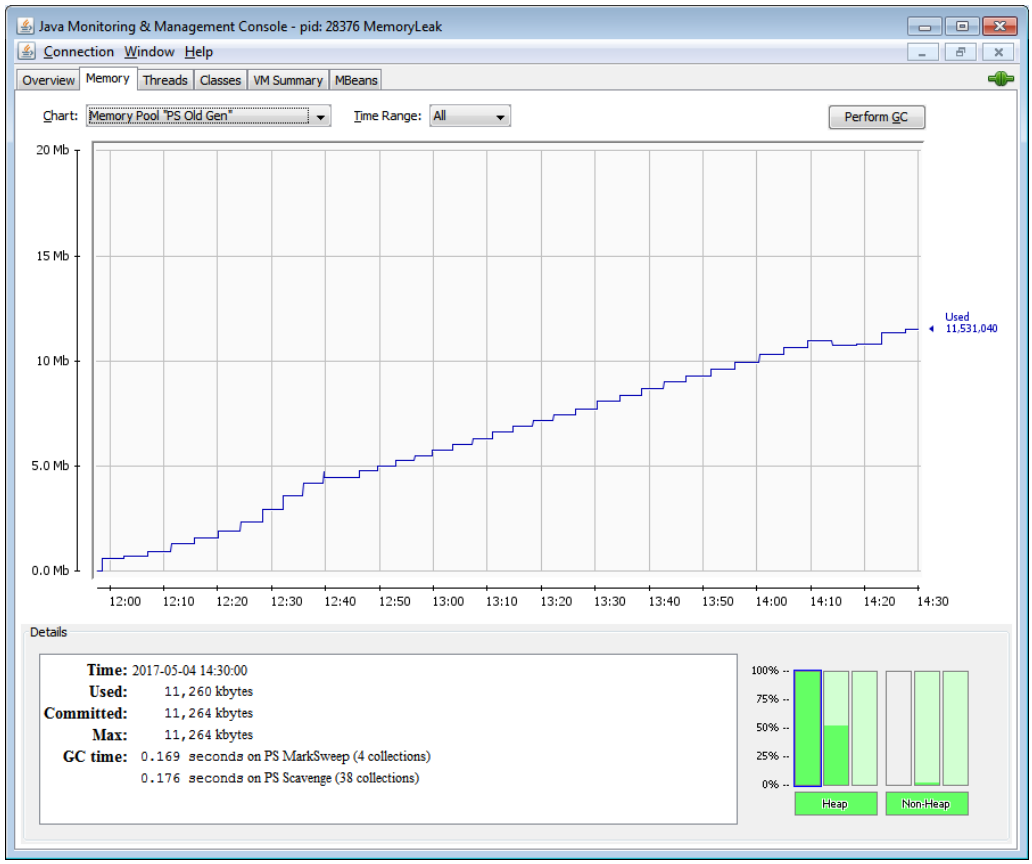


Figure 3-2 JConsole Old Generation



JDK Mission Control

You can detect memory leaks early and prevent `java.lang.OutOfMemoryError` errors using JDK Mission Control (JMC).

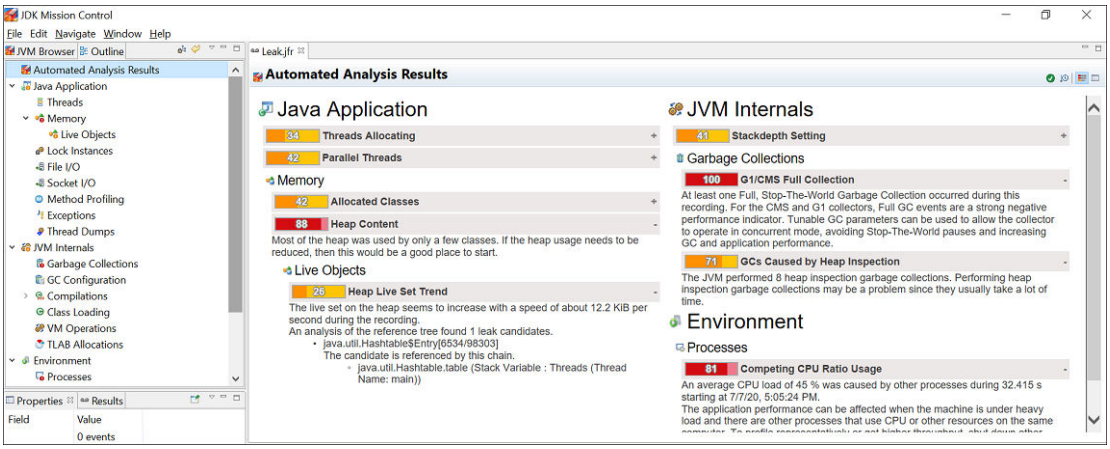
Detecting a slow memory leak can be difficult. A typical symptom could be the application becoming slower after running for a long time due to frequent garbage collections. Eventually, `java.lang.OutOfMemoryError` errors may be seen. However, memory leaks can be detected early, even before such problems 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 being used after a full garbage collection, which collects all the unreachable objects. To inspect the live set, start 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`.

Start 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 an example of a recording, which shows a heap size issue:

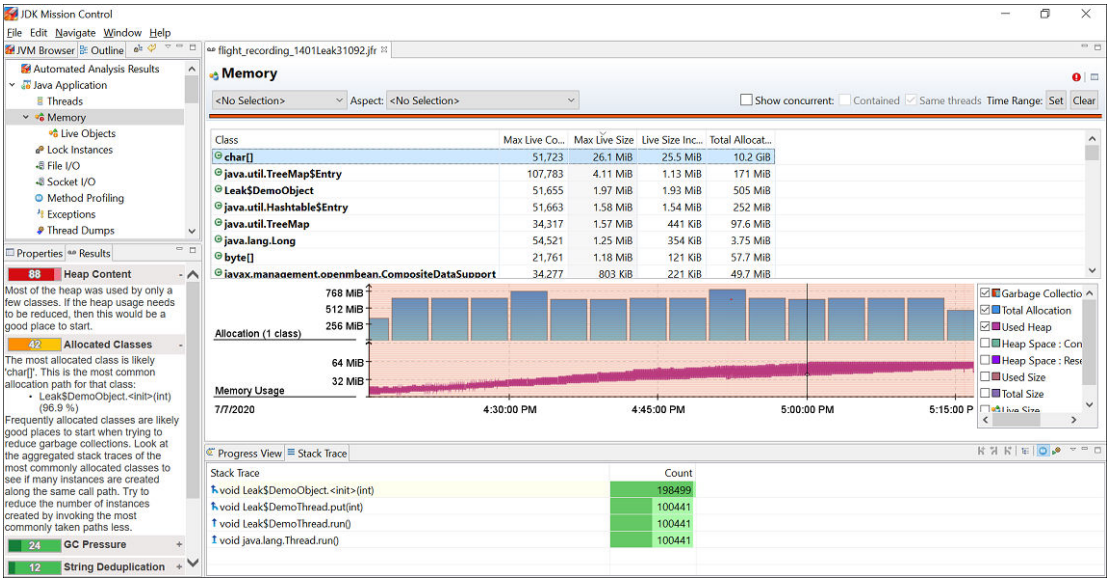
Figure 3-3 Memory Leak - Automated Analysis Page



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 3-4 Memory Leak - Memory Page



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

Garbage Collection logs

Memory usage information can be extracted using GC logs as well. If the GC logs show that the application has performed several full garbage collections attempting to reclaim space in

Old generation or Metaspace, but without any significant gain, this indicates that the application might be suffering from a memory leak problem.

GC logs can be collected using the `-Xlog` command-line Java option. An example is given below:

```
-Xlog:gc*,gc+phases=debug:gc.log
```

This will log messages tagged with at least `gc` using `info` level, and messages tagged with exactly `gc` and `phases` tags using `debug` level to a file called `gc.log`.

Here's an excerpt from a GC log, collected with `-Xlog:gc*`.

```
[4.344s][info][gc,start       ] GC(46) Pause Full (Ergonomics)
[4.344s][info][gc,phases,start] GC(46) Marking Phase
[4.402s][info][gc,phases      ] GC(46) Marking Phase 57.896ms
[4.402s][info][gc,phases,start] GC(46) Summary Phase
[4.402s][info][gc,phases      ] GC(46) Summary Phase 0.023ms
[4.402s][info][gc,phases,start] GC(46) Adjust Roots
[4.402s][info][gc,phases      ] GC(46) Adjust Roots 0.108ms
[4.402s][info][gc,phases,start] GC(46) Compaction Phase
[4.435s][info][gc,phases      ] GC(46) Compaction Phase 33.721ms
[4.436s][info][gc,phases,start] GC(46) Post Compact
[4.436s][info][gc,phases      ] GC(46) Post Compact 0.073ms
[4.436s][info][gc,heap        ] GC(46) PSYoungGen: 12799K(14848K)-
>12799K(14848K) Eden: 12799K(12800K)->12799K(12800K) From: 0K(2048K)-
>0K(2048K)
[4.436s][info][gc,heap        ] GC(46) ParOldGen: 34072K(34304K)-
>34072K(34304K)
[4.436s][info][gc,metaspace   ] GC(46) Metaspace: 149K(384K)-
>149K(384K) NonClass: 145K(256K)->145K(256K) Class: 3K(128K)->3K(128K)
[4.436s][info][gc             ] GC(46) Pause Full (Ergonomics) 45M-
>45M(48M) 92.086ms
[4.436s][info][gc,cpu         ] GC(46) User=0.15s Sys=0.01s Real=0.10s
```

In this example, the Java heap is sized at 48M, and the Full GC was not able to reclaim any space. It is clear that the Old generation is completely full, and the Full GC could not help much. This suggests that either the heap is sized smaller than the application's heap requirements, or there is a memory leak.

Diagnosing Java Memory Leaks

Diagnosing leaks in Java source 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 Java source code.

Diagnostic Data

This section looks at the diagnostic data you can use to troubleshoot memory leaks.

Heap Histograms

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 two 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 --core core_file
  ```

  For example, if you specify the `-XX:+CrashOnOutOfMemoryError` command-line option while running your application, then when a `java.lang.OutOfMemoryError` error is thrown, the JVM will generate a core dump. You can then execute `jhsdb 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:

  num     #instances     #bytes   Class description
  -------------------------------------------------------------------------
  1:            2108     112576    byte[]
  2:             546      66112    java.lang.Class
  3:            1771      56672    java.util.HashMap$Node
  4:             574      53288    java.lang.Object[]
  5:            1860      44640    java.lang.String
  6:             349      40016    java.util.HashMap$Node[]
  7:              16      33920    char[]
  8:             977      31264
  ```

```
java.util.concurrent.ConcurrentHashMap$Node
9:             327      15696    java.util.HashMap
10:            266      13800    java.lang.String[]
11:            485      12880    int[]
:

Total : 14253 633584
Heap traversal took 1.15 seconds.
```

The above example shows that the `java.lang.OutOfMemoryError` error 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.

Heap Dumps

Heap dumps are the most important data to troubleshoot memory leaks. Heap dumps can be collected using `jcmd`, `jmap`, JConsole tools, and the `-XX:+HeapDumpOnOutOfMemoryError` Java option

- You can use the `GC.heap_dump` command with the `jcmd` utility to create a heap dump as shown below:

  ```
  jcmd <process id/main class> GC.heap_dump filename=heapdump.dmp
  ```

- `jmap` with `-dump:format=b` can dump heap from a running process

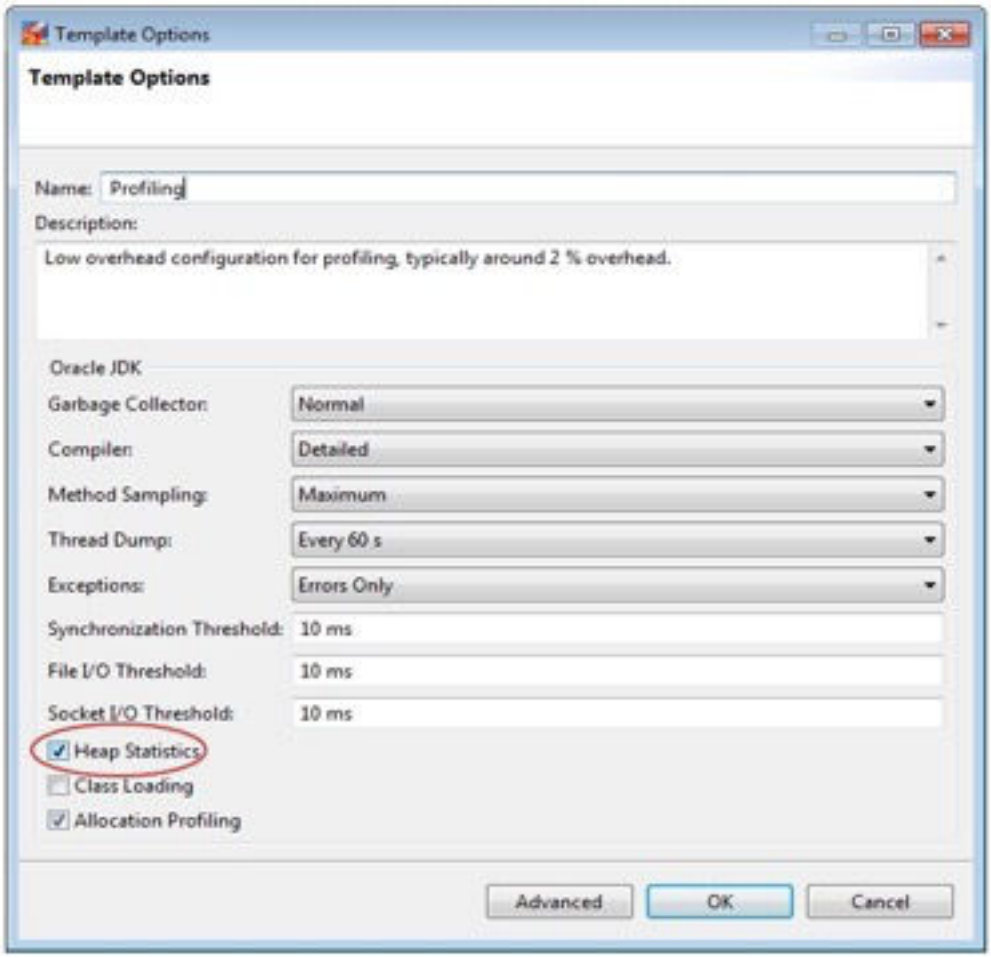
  ```
  jmap -dump:format=b,file=snapshot.jmap <process id>
  ```

- The MBean browser in JConsole makes the `HotSpotDiagnostic` MBean available, and that can be used to create heap dumps for the attached Java process.
- You can use the `-XX:+HeapDumpOnOutOfMemoryError` Java option to dump the Java heap of a process when it fails with a `java.lang.OutOfMemoryError` error.

Java Flight Recordings

Flight Recordings collected with heap statistics enabled can be helpful in troubleshooting a memory leak, showing you the Java objects and the top growers in the Java heap over time. To enable heap statistics, you can use JDK Mission Control (JMC) and enable the ‘Heap Statistics’ by going to ‘Window > Flight Recording Template Manager’ as shown below.

Figure 3-5 Flight Recording Template Manager



Heap Statistics can also be enabled by manually editing the `.jfc` file, and setting `heap-statistics-enabled` to `true`.

```
<event path="vm/gc/detailed/object_count">
    <setting name="enabled" control="heap-statistics-enabled">true</setting>
    <setting name="period">everyChunk</setting>
</event>
```

The flight recordings can then be created using any of the following ways:

- Java Flight Recorder options

  ```
  -XX:StartFlightRecording=delay=20s,duration=60s,name=MyRecording,filename=myrecording.jfr,settings=profile
  ```

- JMC

Class Loader Statistics

Information about class loaders and the number of classes loaded by them can be very helpful in diagnosing memory leaks related to Metaspace and Compressed Class Space.

The class loader statistics information can be collected using `jcmd` and `jmap` tools, as shown in the examples below:

- `jcmd <process id/main name> VM.classloader_stats`
- `jmap -clstats <process id>`

The following is an example of output generated by `jmap`.

```
jmap -clstats 15557
ClassLoader         Parent              CLD*               Classes
ChunkSz   BlockSz  Type
0x0000000800c40ac8  0x0000000800085938  0x00007ff3a18787e0
1       768       184  java.net.URLClassLoader
0x0000000800c40ac8  0x0000000800085938  0x00007ff39f652f90
1       768       184  java.net.URLClassLoader
0x0000000800c40ac8  0x0000000800085938  0x00007ff39f499620
1       768       184  java.net.URLClassLoader
0x0000000800c40ac8  0x0000000800085938  0x00007ff3a194e3a0
1       768       184  java.net.URLClassLoader
0x0000000800c40ac8  0x0000000800085938  0x00007ff39f5aaad0
1       768       184  java.net.URLClassLoader
0x0000000800c40ac8  0x0000000800085938  0x00007ff3a1823d20
1       768       184  java.net.URLClassLoader
0x0000000800c40ac8  0x0000000800085938  0x00007ff3a194cab0
1       768       184  java.net.URLClassLoader
0x0000000800c40ac8  0x0000000800085938  0x00007ff3a1883190
1       768       184  java.net.URLClassLoader
0x0000000800c40ac8  0x0000000800085938  0x00007ff3a191b9c0
1       768       184  java.net.URLClassLoader
0x0000000800c40ac8  0x0000000800085938  0x00007ff3a1914810
1       768       184  java.net.URLClassLoader
0x0000000800c40ac8  0x0000000800085938  0x00007ff3a181c050
1       768       184  java.net.URLClassLoader
0x0000000800c40ac8  0x0000000800085938  0x00007ff39f5c57c0
1       768       184  java.net.URLClassLoader
0x0000000800c40ac8  0x0000000800085938  0x00007ff39f6774d0
1       768       184  java.net.URLClassLoader
0x0000000800c40ac8  0x0000000800085938  0x00007ff3a18574b0
1       768       184  java.net.URLClassLoader
0x0000000800c40ac8  0x0000000800085938  0x00007ff3a1803500
1       768       184  java.net.URLClassLoader
…
Total = 1105                                        1757   1404032
308858
ChunkSz: Total size of all allocated metaspace chunks
BlockSz: Total size of all allocated metaspace blocks (each chunk has
several blocks)
```

Analysis Tools

This section explores the analysis tools you can use to diagnose memory leaks, including those that can analyze diagnostic data described above.

Heap Dump Analysis Tools

There are many third-party tools available for heap dump analysis. The Eclipse Memory Analyzer Tool (MAT) and YourKit are two examples of commercial tools with memory debugging capabilities. There are many others, and no specific product is recommended.

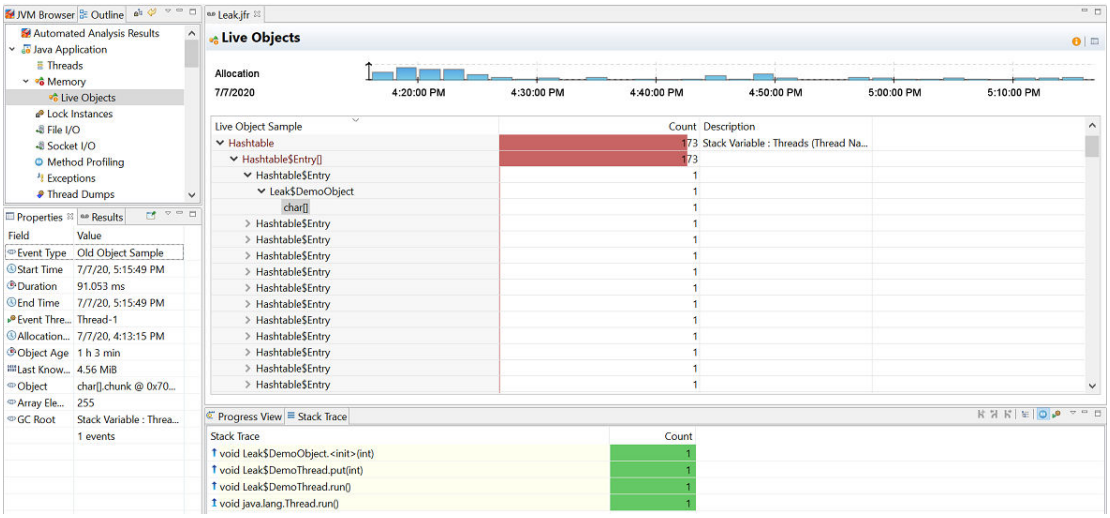
JDK Mission Control (JMC)

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

This section describes how to debug a memory leak by analyzing a flight recording in JMC.

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

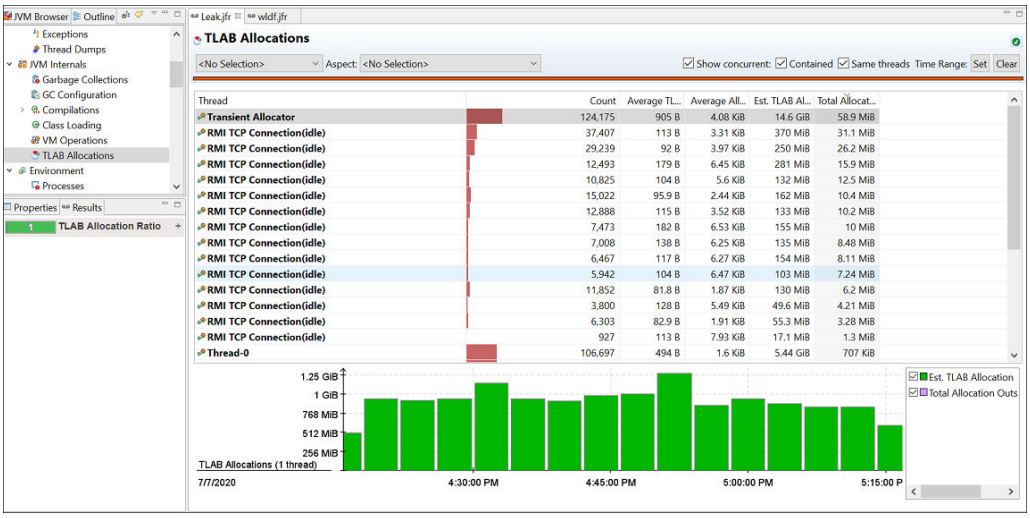
Figure 3-6 Memory Leak - Live Objects Page



You can observe that most of the live objects being tracked are held by `Leak$DemoThread`, which in turn holds 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, and 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 recording, showing TLAB allocations.

Figure 3-7 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 `java.lang.OutOfMemoryError` error, a recording with the prefix `hs_oom_pid` 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 `jcmd` 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
  ]
  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
  ]
  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 allocation-intensive applications, but is typically OK when debugging. Overhead is usually less than 2%.

Setting `path-to-gc-roots=true` 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.

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.

Diagnosing Native Memory Leaks

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.

- Tracking All Memory Allocation and Free Calls
- Tracking All Memory Allocations in the JNI Library

Tracking All Memory Allocation and Free Calls

A very common practice is to track all allocation and free calls of 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 such as IBM Rational Purify 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 uninitialized memory or accesses to freed memory.

Not all these types of tools will work with Java applications that use native code, and usually these tools are platform-specific. Because the JVM 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 JVM 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.

Native memory leaks can result from native allocations performed either internally by the JVM, or from outside the JVM. The following two sections discuss in detail how both of these memory leaks can be diagnosed.

Native Memory Leaks for Allocations performed by the JVM

The JVM has a powerful tool called Native Memory Tracking (NMT) that tracks native memory allocations performed internally by the JVM. Note that this tool cannot track native memory allocated outside of the JVM, for example by JNI code.

Here is how this tool can be used:

- Enable NMT in the process that you want to monitor by using the Java option `NativeMemoryTracking`. The output level of the tracking can be set to a `summary` or `detail` level as shown below:

```
-XX:NativeMemoryTracking=summary
-XX:NativeMemoryTracking=detail
```

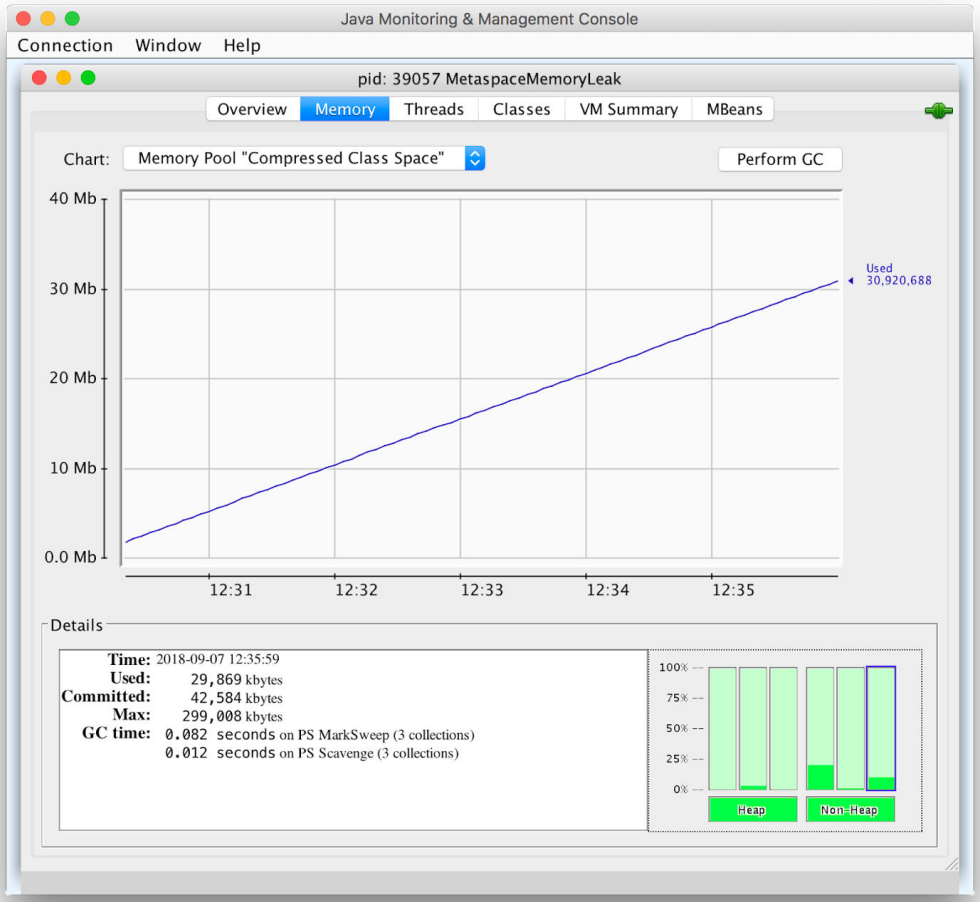
- The **jcmd** tool can then be used to attach to the NMT-enabled process, and obtain its native memory usage details. It is also possible to collect a baseline of memory usage and then collect the difference in usage against that baseline.

```
jcmd <process id/main class> VM.native_memory
jcmd <process id/main class> VM.native_memory baseline
jcmd <process id/main class> VM.native_memory detail.diff/
summary.diff
```

> **Note:**
>
> Enabling NMT can result in a performance drop of around 5 to 10 percent. Therefore, it should be enabled in production systems cautiously. Additionally, the native memory used by NMT is tracked by the tool itself.

In the following example, the JConsole screenshot shows that the usage of 'Compressed Class Space' is growing at a constant rate over time.

Figure 3-8 JConsole Compressed Class Space



To diagnose this usage growth, the Java process can be tracked using the NMT. Collecting a baseline and a `summary.diff` output shows that the class space usage is increasing dramatically due to the corresponding increase in the number of loaded classes.

```
bash-3.2$ jcmd 39057 VM.native_memory summary.diff
39057:
Native Memory Tracking:
```

```
Total: reserved=5761678KB +52943KB, committed=472350KB +104143KB
-           Java Heap (reserved=4194304KB, committed=163328KB +7680KB)
                            (mmap: reserved=4194304KB,
committed=163328KB +7680KB)

-              Class (reserved=1118333KB +47579KB, committed=117949KB
+89963KB)
                   (classes #68532 +58527)
                   (malloc=8317KB +2523KB #5461 +3371)
                   (mmap: reserved=1110016KB +45056KB,
committed=109632KB +87440KB)

-              Thread (reserved=21594KB -2057KB, committed=21594KB
-2057KB)
                    (thread #22 -2)
                    (stack: reserved=21504KB -2048KB, committed=21504KB
-2048KB)
                    (malloc=65KB -6KB #111 -10)
                    (arena=25KB -2 #42 -4)

-              Code (reserved=250400KB +244KB, committed=5612KB
+1348KB)
                    (malloc=800KB +244KB #1498 +234)
                    (mmap: reserved=249600KB, committed=4812KB +1104KB)

-              GC (reserved=159039KB +18KB, committed=145859KB +50KB)
                    (malloc=5795KB +18KB #856 +590)
                    (mmap: reserved=153244KB, committed=140064KB +32KB)

-              Compiler (reserved=153KB, committed=153KB)
                    (malloc=22KB #72 -2)
                    (arena=131KB #3)

-              Internal (reserved=13537KB +6949KB, committed=13537KB
+6949KB)
                     (malloc=13505KB +6949KB #70630 +59119)
                     (mmap: reserved=32KB, committed=32KB)

-              Symbol (reserved=2715KB +9KB, committed=2715KB +9KB)
                      (malloc=1461KB +9KB #702 +29)
                      (arena=1255KB #1)

-       Native Memory Tracking (reserved=1416KB +1031KB,
committed=1416KB +1031KB)
                       (malloc=140KB +34KB #2197 +518)
                       (tracking overhead=1275KB +997KB)

-               Arena Chunk (reserved=186KB -832KB, committed=186KB
-832KB)
                            (malloc=186KB -832KB)
```

Native Memory Leaks from Outside the JVM

For native memory leaks originating outside the JVM, you can use platform native or other third-party tools for their detection and troubleshooting. Here is a list of some of the tools that you can find useful in troubleshooting native memory leaks caused by allocations performed outside the JVM.

- Valgrind
- Purify available on both UNIX platforms and Windows
- Using Crash Dump or Core files
- On Windows, search Microsoft Docs for debug support. The Microsoft C++ compiler has the `/Md` and `/Mdd` compiler options that will automatically include extra support for tracking memory allocation. User-Mode Dump Heap (UMDH) is also helpful in tracking memory allocations.
- Linux systems have tools such as `mtrace` and `libnjamd` to help in dealing with allocation tracking.

The following section takes a closer look at two of these tools.

Valgrind

Valgrind can be used to diagnose native memory leaks on Linux. To monitor a Java process using Valgrind, it can be launched as follows:

```
$ valgrind --leak-check=full --show-leak-kinds=all --
suppressions=suppression_file --log-file=valgrind_with_suppression.log -v
java <Java Class>
```

A suppression file can be supplied to valgrind with `--log-file` option in order for it to not consider the JVM internal allocations (such as the Java heap allocation) as potential memory leaks, otherwise it becomes very difficult to parse through the verbose output and manually look for relevant leak reports.

The following are the contents of a sample suppression file:

```
{
   name
   Memcheck:Leak
   fun:*alloc
   ...
   obj:/opt/java/jdk16/jre/lib/amd64/server/libjvm.so
   ...
}
```

With the above command, and with suppressions in place, Valgrind writes the identified leaks in the specified log file. An example is shown below:

```
==5200== 88 bytes in 1 blocks are still reachable in loss record 461 of
18,861
==5200==    at 0x4C2FB55: calloc (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==5200==    by 0x7DCB156: Java_java_util_zip_Deflater_init
```

```
(in /opt/jdk/ /jre/lib/amd64/libzip.so)
==5200==    by 0x80F54FC: ???
==5200==    by 0x8105F87: ???
==5200==    by 0xFFFFFFFE: ???
==5200==    by 0xEC67F74F: ???
==5200==    by 0xC241B03F: ???
==5200==    by 0xEC67D767: ???
==5200==    by 0x413F96F: ???
==5200==    by 0x8101E7B: ???
==5200==
==5200== 88 bytes in 1 blocks are still reachable in loss record 462
of 18,861
==5200==    at 0x4C2FB55: calloc (in /usr/lib/valgrind/
vgpreload_memcheck-amd64-linux.so)
==5200==    by 0x7DCB156: Java_java_util_zip_Deflater_init
(in /opt/jdk/jre/lib/amd64/libzip.so)
==5200==    by 0x80F54FC: ???
==5200==    by 0x8105F87: ???
==5200==    by 0xFFFFFFFE: ???
==5200==    by 0xEC67FF3F: ???
==5200==    by 0xC241B03F: ???
==5200==    by 0xEC630EB7: ???
==5200==    by 0x413F96F: ???
==5200==    by 0x8101E7B: ???
==5200==    by 0x41: ???
==5200==    by 0x19EAE47F: ???
```

In the above output, Valgrind correctly reports that there are allocations leaking from *Java_java_util_zip_Deflater_init* native method.

> **Note:**
>
> Using Valgrind may have a negative impact on the performance of the monitored application.

Crash Dump or Core files

On UNIX platforms, the `pmap` tool is helpful in identifying the memory blocks that might be changing/growing in size over time. Once you have identified the growing memory blocks or sections, you can examine the corresponding crash dump or core file(s) to look at those memory blocks. The values and contents at those locations can provide some valuable clues, which can help tie them back to the source code responsible for the allocations in those memory blocks.

```
$ diff pmap.15767.1 pmap.15767.3
69,70c69,70
< 00007f6d68000000 17036K rw--- [ anon ]
< 00007f6d690a3000 4850K ----- [ anon ]
---
> 00007f6d68000000 63816K rw--- [ anon ]
> 00007f6d690a3000 65536K ----- [ anon ]
```

From the previous `pmap` output, we can see that the memory block at 00007f6d690a3000 is growing between the two memory snapshots of the process. Using a core file collected from the process, we can examine the contents of this memory block.

```
$ gdb `which java` core.15767
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
...
(gdb) x/100s 0x00007f6d690a3000
0x7f6d690a3000: "mory Leak "
0x7f6d690a300c: "Alert: JNI Memory Leak "
0x7f6d690a3025: "Alert: JNI Memory Leak "
0x7f6d690a303e: "Alert: JNI Memory Leak "
0x7f6d690a3057: "Alert: JNI Memory Leak "
0x7f6d690a3070: "Alert: JNI Memory Leak "
```

The above shows that there is a repeating string, "Alert: JNI Memory Leak", present in that memory block. Searching in the source code for the string or contents found in the relevant memory block can lead us to the culprit in the code. Here is the code used for this example, where these allocations are performed in JNI code and are not being released.

```
JNIEXPORT void JNICALL Java_JNINativeLeak_allocateMemory
(JNIEnv *env, jobject obj, jint size) {
    char* bytes = (char*) malloc(size);
    printf("Allocated %d bytes at %p \n", size, (void*)bytes);
    for (int i=0; i<40; i++) {
        strcpy(bytes+i*25, "Alert: JNI Memory Leak ");
    }
}
```

Hence, the pmap tool and core files can help in getting to the root of native memory leaks caused by allocations performed outside of the JVM.

Tracking 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)->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.

Monitoring the Objects Pending Finalization

Different commands and options available to monitor objects pending finalization.

When the `java.lang.OutOfMemoryError` error is thrown with the "Java heap space" detail message, the cause can be the 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 heavily on finalization.
- On Linux, 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.

Troubleshooting a Crash Instead of a java.lang.OutOfMemoryError error

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 an 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.

4

Troubleshoot Performance Issues Using Flight Recorder

This chapter describes how to identify performance issues with a Java application and debug these issues using flight recordings.
To learn more about creating a recording with Flight Recorder in Java Mission Control (JMC), see Start 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

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, 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. The Heap Statistics feature is useful when debugging memory leaks or when investigating the live set of the application. For more information, see The jfr tool.

> **Note:**
>
> For performance profiling use cases, heap statistics may not be necessary.

Find Bottlenecks

Different applications have different bottlenecks. Waiting for I/O or networking, synchronization between threads, CPU usage or garbage collection times can cause bottlenecks in an application. It is possible that an application has more than one bottleneck.

Topics:

Use JDK Mission Control to Find Bottlenecks

You can use JMC to find application bottlenecks.

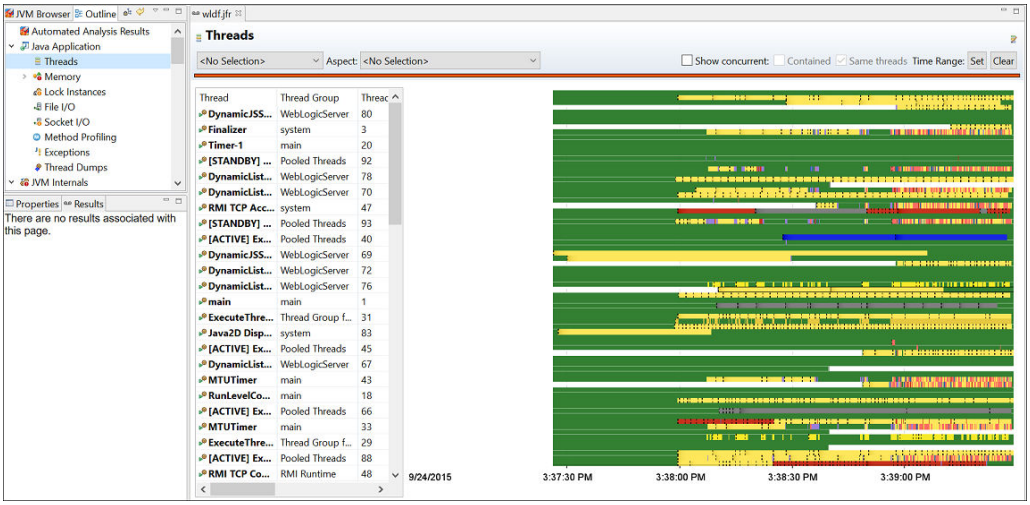
One way to find out the application bottlenecks is to analyze the **Automated Analysis Results** page. This page provides comprehensive automatic analysis of flight recording data.

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 the jfr Tool to Find Bottlenecks

Different applications have different bottlenecks. For some applications, a bottleneck may be waiting for I/O or networking, it may be synchronization between threads, or it may be actual CPU usage. For others, a bottleneck may be garbage collection times. It is possible that an application has more than one bottleneck.

One way to find the application bottlenecks is to look at the following events in your flight recording. Make sure that all of these events are enabled in the recording template that you are using:

- `jdk.FileRead`
- `jdk.FileWrite`
- `jdk.SocketRead`
- `jdk.SocketWrite`
- `jdk.JavaErrorThrow`
- `jdk.JavaExceptionThrow`
- `jdk.JavaMonitorEnter`

- `jdk.JavaMonitorWait`
- `jdk.ThreadStart`
- `jdk.ThreadEnd`
- `jdk.ThreadSleep`
- `jdk.ThreadPark`

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.

Use the `jfr` tool to print the events that were recorded and look for the following information:

- `jdk.JavaMonitorWait` events show how much time a thread spends waiting for a monitor.
- `jdk.ThreadSleep` and `jdk.ThreadPark` events show when a thread is sleeping or parked.
- Read and write events show how much time is spent in I/O.

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. 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 with. 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. Each of these bottlenecks can be further investigated within the flight recording.

> **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.) To summarize, 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.

Garbage Collection Performance

Flight recordings can help you diagnose garbage collection issues in Java application.

Topics:

Use JDK Mission Control to Debug Garbage Collection Issues

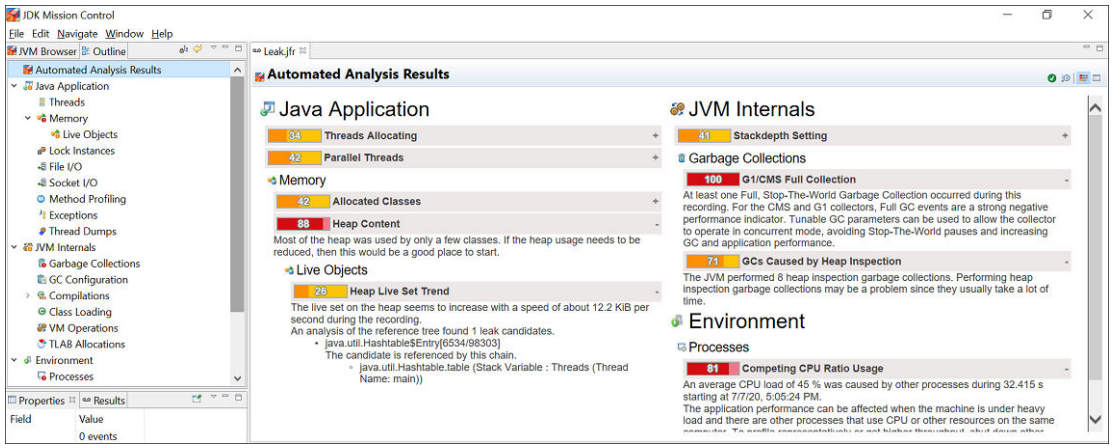
You can use JMC to debug garbage collections (GC) issues.

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.

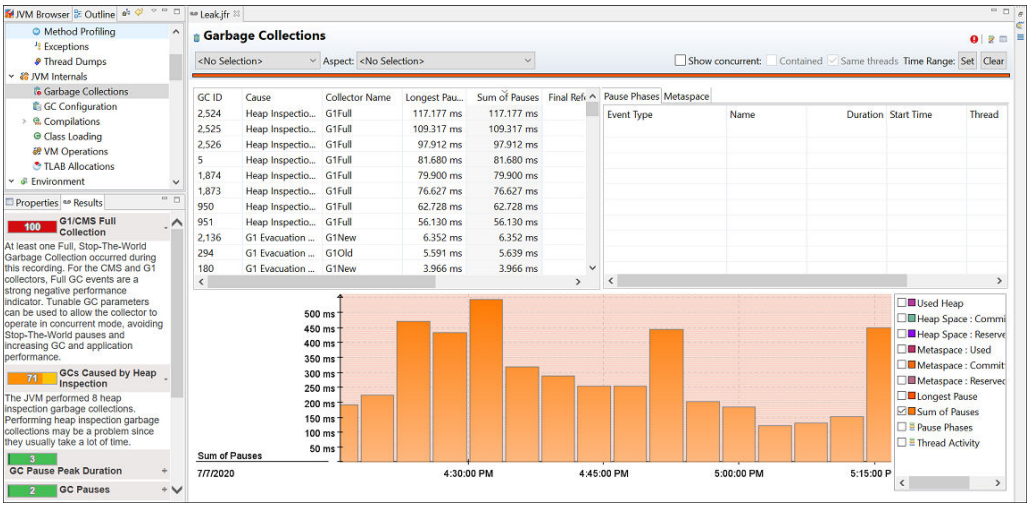
Figure 4-2 Automated Analysis Results - Garbage Collections



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 *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide*.

In addition, you may also need to fix your application so that it makes less use of finalizers or semireferences. See Monitoring 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 The jfr tool. 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 in a Thread Local Area Buffer (TLAB). TLAB is a small memory area where new objects are allocated. Once a TLAB is full, the thread gets a new one. Larger objects are allocated outside a TLAB. Often, the majority of allocations happen inside a TLAB. 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 *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide* for more information about GC performance.

Use the jfr Tool to Debug Garbage Collection Issues

Recordings from Flight Recorder can help diagnose Java application issues with garbage collections.

Tuning the HotSpot Garbage Collector can have a big effect on performance. See Introduction to Garbage Collection Tuning in *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide* for more information.

To investigate garbage collection issues, take a profiling flight recording of your application while it is running. Do not include the heap statistics, because that triggers extra old collections. To get a good sample, take a longer recording, for example, 1 hour.

Use the `jfr` tool to print the `jdk.GCPhasePause` events that were recorded. The following example shows the information contained in the event:

```
c:\Program Files\Java\jdk-15\bin>jfr print --events jdk.GCPhasePause \
   gctest.jfr
jdk.GCPhasePause {
  startTime = 11:19:13.779
  duration = 3.419 ms
  gcId = 1
  name = "GC Pause"
  eventThread = "VM Thread" (osThreadId = 17528)
}
```

Using the information from the `jdk.GCPhasePause` events, you can calculate the average sum of pauses for each GC, the maximum sum of pauses, and the total pause time. 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 versus throughput performance. See Behavior-Based Tuning in *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide*.

In addition, you may also need to fix your application so that it makes less use of finalizers or semireferences. See Monitoring 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.

When the application spends too much time paused, there are different ways to work around that:

- Increase the Java heap size. The bigger the Java heap, the longer time it is between GCs. Watch out for any memory leaks in the Java application, because that may cause more and more frequent GCs until an `OutOfMemoryError` is thrown. For more information, see The jfr tool.
- To reduce the number of GCs, allocate fewer temporary objects. Small objects are allocated in a Thread Local Area Buffer (TLAB). TLAB is a small memory area where new objects are allocated. Once a TLAB is full, the thread gets a new one. Larger objects are allocated outside a TLAB. Often, the majority of allocations happen inside a TLAB. The `jdk.ObjectAllocationInNewTLAB` and `jdk.ObjectAllocationOutsideTLAB` events provide information about the allocation of temporary objects.
- To reduce the need of GCs, decrease the allocation rate. The `jdk.ThreadAllocationStatistics` event provides information about the allocations per thread.

Some other settings may also increase GC performance of the Java application. See Garbage-First Garbage Collection in *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide* for more information about GC performance.

Synchronization Performance

Java applications encounter synchronization issues when the application threads spend a lot of time waiting to enter a monitor.

Topics:

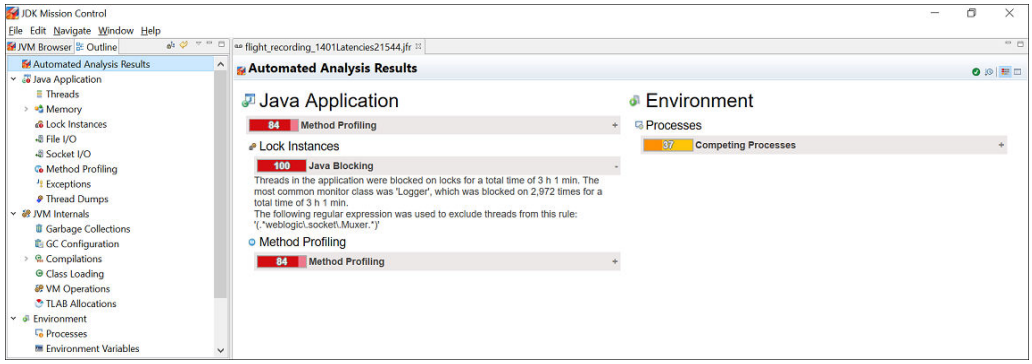
- Use JDK Mission Control to Debug Synchronization Issues
- Use jdk.JavaMonitorWait Events to Debug Synchronization Issues

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.

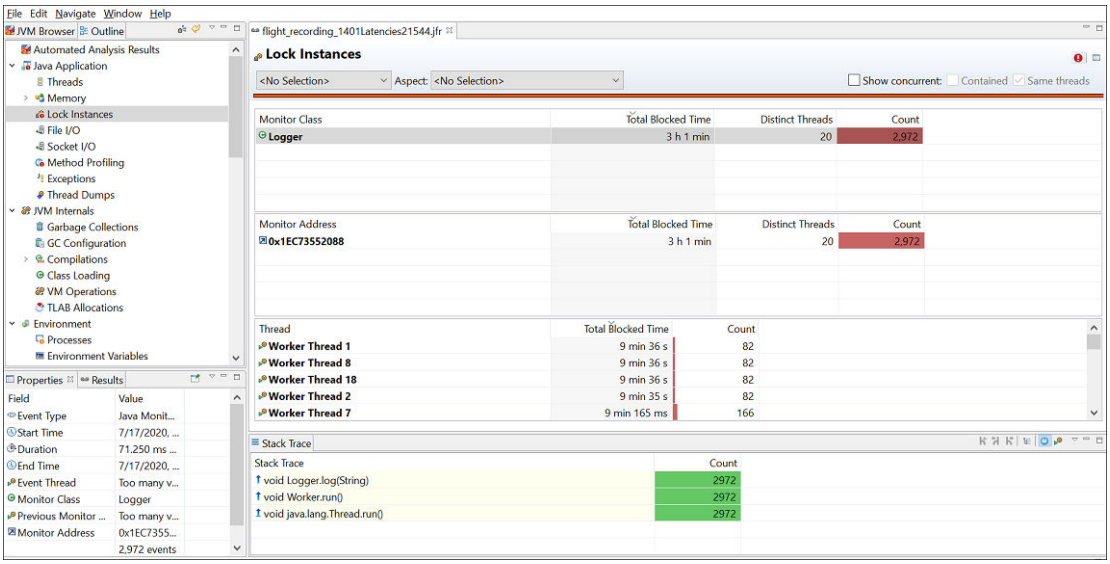
Figure 4-4 Synchronization Issue - Automated Analysis Results Page



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.

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.JavaMonitorWait Events to Debug Synchronization Issues

To debug Java Application synchronization issues, which is where the application threads spend a lot of time waiting to enter a monitor, look at the `jdk.JavaMonitorWait` events in a recording from Flight Recorder.

Look at the locks that are contended the most and the stack trace of the threads waiting to acquire the lock. Typically, look for contention that you did not think would be an issue. Logging is a common area that can be an unexpected bottleneck in some applications.

When you see performance degradation after a program update or at any specific times in the Java application, take a flight recording when things are good, and take another one when things are bad to look for a synchronization site that increases a lot.

> **Note:**
>
> By default, contention events with a duration longer than 20 ms are recorded. This threshold can be modified when starting the flight recording. Shorter thresholds give more events and also potentially more overhead. If you believe contention is an issue, then you could take a shorter recording with a very low threshold of only a few milliseconds. When this is done on a live application, make sure to start with a very short recording, and monitor the performance overhead.

I/O Performance

Flight recordings can help you diagnose I/O performance issues in Java application.

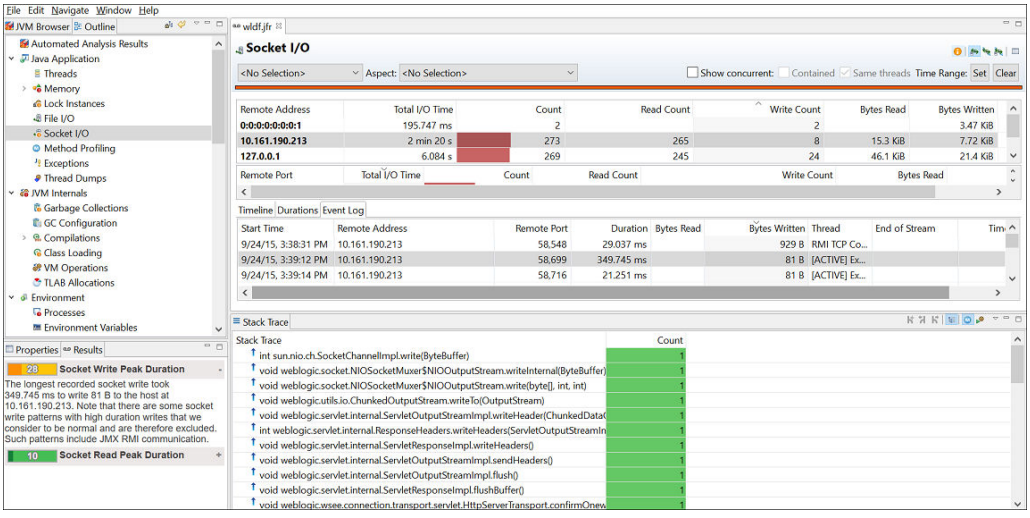
Topics:

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.

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 the Socket Read and Write Events to Debug I/O Issues

When a Java application spends a lot of time reading or writing sockets or files, then I/O or networking may be the bottleneck. Recordings from Flight Recorder can help identify problem areas.

To diagnose I/O issues in applications, look at the following events in your flight recording. Make sure that all of these events are enabled in the recording template that you are using:

- `jdk.SocketWrite`
- `jdk.SocketRead`
- `jdk.FileWrite`
- `jdk.FileRead`

Use the socket read and write information in your flight recording to calculate the number of reads from a specific remote address, the total number of bytes read, and the total time spent waiting. Look at each event to analyze the time spent and data read.

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

By default, only events with a duration longer than 20 ms are recorded. When starting a flight recording, you can lower the file I/O threshold or the socket I/O threshold to gather more data, potentially with a higher performance effect.

Code Execution Performance

Flight recordings can help you diagnose code execution performance issues in Java application.

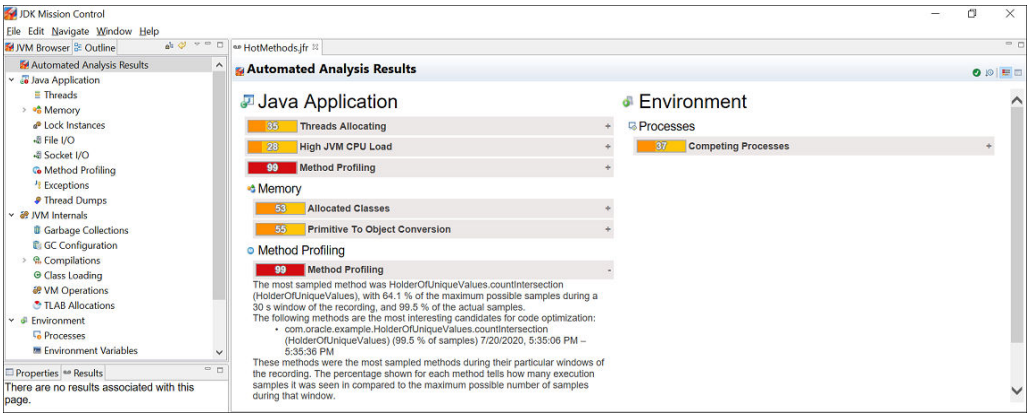
Topics:

Use JDK Mission Control to Monitor Code Execution Performance

You can use JMC to monitor the code execution performance.

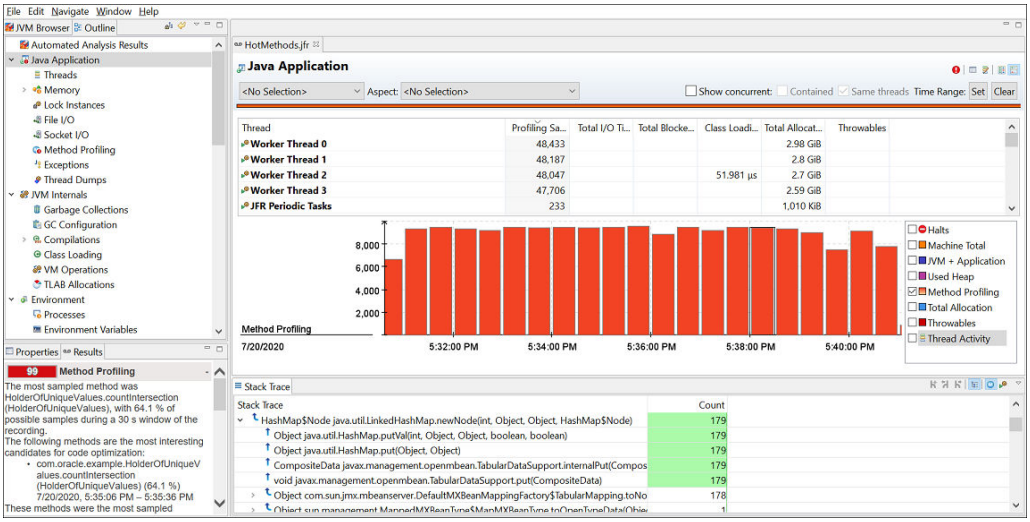
When there are not a lot of Java Application events, it could be that the main bottleneck of your application is the running code. In such scenarios, look at the **Method Profiling** section of the **Automated Analysis Results** page. Here is a sample figure of a recording, which indicates that there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

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.

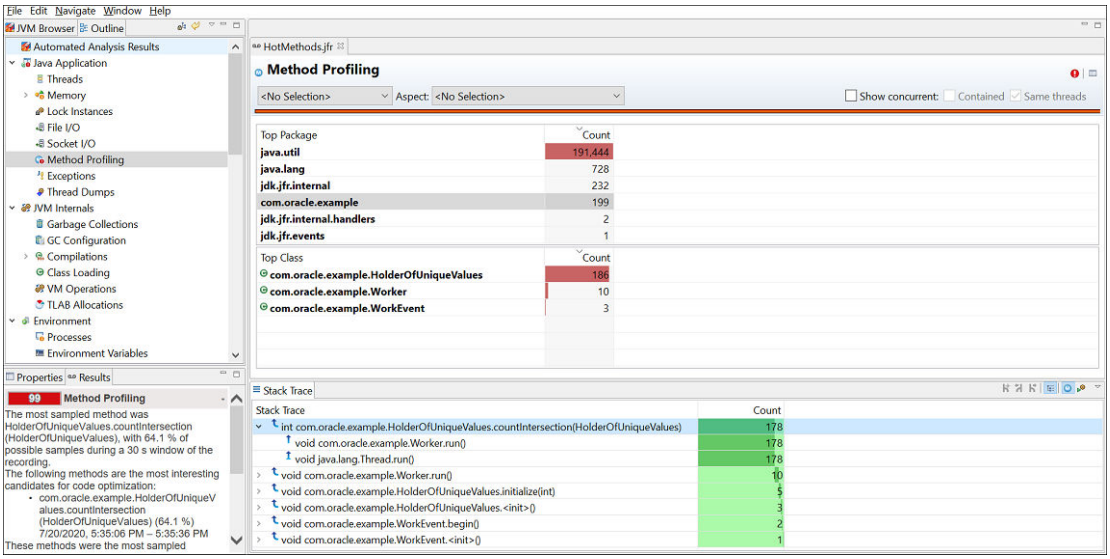
Figure 4-8 Code Execution Performance - Java Application



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.

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.

Use jdk.CPULoad and jdk.ThreadCPULoad Events to Monitor 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. Recordings from Flight Recorder can help identify problem areas.

Look at the `jdk.CPULoad` events and review the CPU usage over time. This shows the CPU usage of the JVM being recorded and the total CPU usage on the machine. If the JVM CPU usage is low, but the CPU usage of the machine is high, then some other application is likely taking a lot of CPU. In that case, look at the other applications running on the system using OS tools such as **Top** or the task manager to find out which processes are using a lot of CPU.

In case your application is using a lot of CPU time, look at `jdk.ThreadCPULoad` events and identify the threads that use the most CPU time. This information is based on method sampling, so it may not be 100% accurate if the sample count is low. When a recording is running, the JVM samples the threads. By default, a continuous recording does only some method sampling, while a profiling recording does as much as possible. The method sampling gathers data from only those threads running code. The threads waiting for I/O, sleeping, waiting for locks, and so on are not sampled. Therefore, threads with a lot of method samples are the ones using the most CPU time; however, how much CPU is used by each thread is not known.

The **Hot Methods** tab in the **Code** tab group helps find out where your application spends most of the execution time. This tab shows all the samples grouped by top method in the stack. Use the **Call Tree** tab to start with the lowest method in the stack traces and then move upward. starts with `Thread.run`, and then looks at the calls that have been most sampled.

5
Troubleshoot Security APIs

To learn more about troubleshooting security APIs, see Troubleshooting Security.

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.

6
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

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

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 JDK (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 JDK libraries in the `<java-home>/lib` or `<java-home>/bin` directories of the JDK 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 `java.lang.StackOverflowError` 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]
#

---------------  T H R E A D  ---------------

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 command-line 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 the `-XX:CompileCommand` option in Advanced JIT Compiler Options for java in the *Java Development Kit Tool Specifications*.

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)
:
---------------  T H R E A D  ---------------

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 `exclude`*`classmethod`*, 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.

> **Note:**
>
> The file name separator is a dot, not a slash.

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:   00000000 00000000 00000001 00000001
0x02aefc6c:   00000000 00000000 00000000 08072a9e
0x02aefc7c:   00000000 00000000 00000000 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:   0f 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]
V  [jvm.dll+0x6e4c5]
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 Linux operating system. 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 Linux operating system, 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 JDK 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.

7

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.

This chapter contains the following sections:

Diagnose a Loop Process

If a VM process appears to be looping, try to get a thread dump. A thread dump will often make 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 Linux) or the Control+Break key combination (on Windows) to cause the HotSpot VM to print a thread dump, including thread state. On 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. 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 Linux), or Control+Break (on Windows) to cause the HotSpot VM to print a thread dump. On the Linux operating system, 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.

The following example shows the stack trace for this situation.

```
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$SharedOwnerFrame.windowClosed(SwingUtilities.java:
1722)
        at java.awt.Window.processWindowEvent(Window.java:1173)
        at javax.swing.JDialog.processWindowEvent(JDialog.java:407)
        at java.awt.Window.processEvent(Window.java:1128)
        at java.awt.Component.dispatchEventImpl(Component.java:3922)
        at java.awt.Container.dispatchEventImpl(Container.java:2009)
        at java.awt.Window.dispatchEventImpl(Window.java:1746)
        at java.awt.Component.dispatchEvent(Component.java:3770)
        at java.awt.EventQueue.dispatchEvent(EventQueue.java:463)
        at
java.awt.EventDispatchThread.pumpOneEventForHierarchy(EventDispatchThread.jav
a:214)
        at
java.awt.EventDispatchThread.pumpEventsForHierarchy(EventDispatchThread.java:
163)
        at
java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:157)
        at
java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:149)
        at java.awt.EventDispatchThread.run(EventDispatchThread.java:110)
"main":
        at java.awt.Window.getOwnedWindows(Window.java:844)
        - waiting to lock <0xf0c41ec8> (a java.util.Vector)
        at
javax.swing.SwingUtilities$SharedOwnerFrame.installListeners(SwingUtilities.j
ava:1697)
        at
javax.swing.SwingUtilities$SharedOwnerFrame.addNotify(SwingUtilities.java:169
0)
        at java.awt.Dialog.addNotify(Dialog.java:370)
        - locked <0xf0c30560> (a java.awt.Component$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 `Object.wait()`. The caller frame in the stack trace indicates the class and method that is invoking the `wait()` 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 the `jstack` mode of jhsdb 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 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 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.

8

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 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 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 8-2 lists the signals that are currently used on the 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 Linux operating system. 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 `catch` 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 `__try` and `__finally` 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.

The 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.

Signal Chaining

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

Handle Exceptions Using the Java HotSpot VM

The HotSpot VM installs a top-level exception handler during initialization using the `AddVectoredExceptionHandler`API 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 8-1.

Table 8-1 Console Events

Console Event	Signal	Usage
`CTRL_C_EVENT`	`SIGINT`	This event and signal is used to terminate a process. (Optional)
`CTRL_CLOSE_EVENTCTRL_LOGOFF_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)

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 Linux and macOS

This topic describes a list of signals that are used on Linux and macOS

Table 8-2 Signals Used on 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)
SIGUSR2	This signal is used internally on Linux and macOS.
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.

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 the Java Runtime Environment (JRE) and troubleshooting techniques to resolve these issues.

9

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 Java Runtime Environment (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 Java Time Zone Updater Tool

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 Java Time Zone Updater Tool

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 a Java runtime for Linux (x64), Microsoft Windows (x64), and macOS (x64). 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 Java runtime determines the default time zone on 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

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.
2. 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 Pipeline Rendering and Properties

  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.

- 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.

10

Introduction to Client Issues

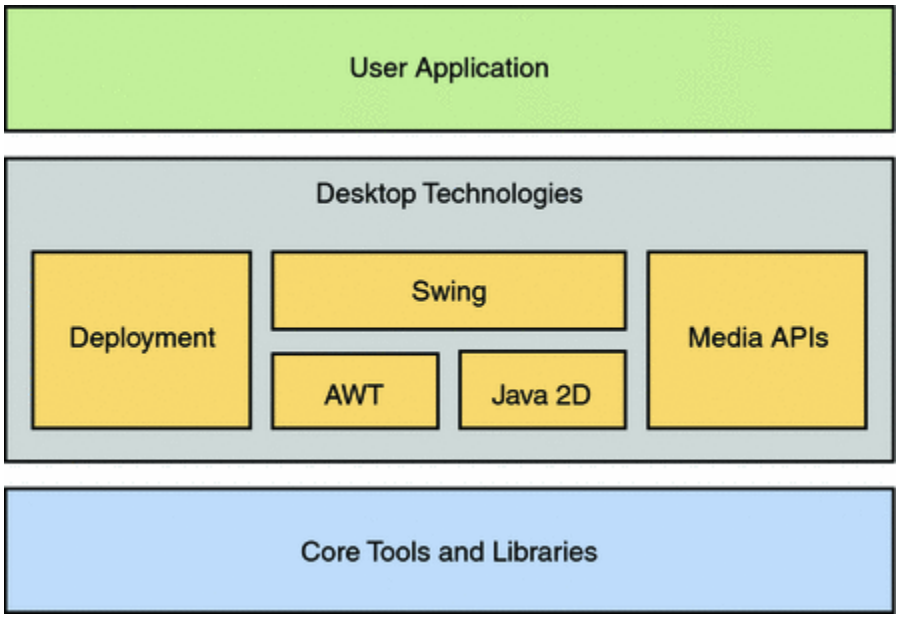
This chapter explains how the different Java SE Desktop technologies interact with each other. In addition, the chapter helps you to pinpoint the technology from which you might start troubleshooting your problem and provides general troubleshooting tips.
This chapter contains the following sections:

Java SE Desktop Technologies

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 10-1.

Figure 10-1 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 Linux operating system.

- **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

This section provides 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 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 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

This section provides a list of basic tools that can help you 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

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.
2. 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 the Linux operating system:

     ```
     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 the Linux operating system).
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 the Linux operating system:

```
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 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.

11
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

This section describes 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 Linux operating system (which sends the SIGQUIT signal).

To trace X11 errors on the Linux operating system, set the `sun.awt.noisyerrorhandler` system property to `true`.

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

Some issues related to handling key events 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.

Modality Issues

This section provides information about issues related to using modality.

The section addresses the following issues.

- **UNIX window managers:**

  Many of the modality improvements are unavailable in some Linux environments, for example, when using Common Desktop Environment (CDE) window managers. 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. 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 the Linux operating system, 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 Linux platform, 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)`.

3. **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 most-recent 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-and-drop 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` or `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.

     ```
     public void dragGestureRecognized(DragGestureEvent dge) {
                 dge.startDrag(null, new
     StringSelection("SomeTransferedText"));

     dge.getDragSource().addDragSourceListener(target);
             }
     ```

   - 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 Linux operating system.**

   Solution: This is a native limitation of X11. On the Linux operating system, 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
```

12

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 troubleshoot 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:

- Linux: X11 Pipeline is the default for the Linux operating system.
- 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 Linux and Windows is an alternative on the Linux operating system, as well as Windows.

Linux: X11 Pipeline

On the Linux operating system, 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

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:**

  It is sometimes necessary to increase the amount of shared memory available to the system (and to X server in particular) because the default is too low, resulting in poor rendering performance. Increasing the amount of shared memory and shared memory segments can result in better performance.

  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`.

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 Linux and Windows

The OpenGL pipeline is available on Linux and Windows.

This alternate pipeline uses the hardware-accelerated, cross-platform OpenGL API when rendering to `VolatileImages`, to back buffers 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 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 Linux operating system 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 `J2D_TRACE_LEVEL` 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.

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 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.

Diagnose Rendering and Performance Issues

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.

The `GL_EXT_framebuffer_object` extension provides better performance for rendering and reduced VRAM consumption when using `VolatileImages`. This "FBO" code path 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" code path 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` code path.

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 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 machine. Drivers can be downloaded from your graphics card manufacturer's web site, as shown in the following table.

Manufacturer	Platforms	Cards Known to Work
AMD/ATI	Linux, Windows	Radeon 8500 and later, FireGL series
Nvidia	Linux, Windows	GeForce 2 series and later, Quadro FX series and later
Xi Graphics	Linux	Various (check with Xi Graphics)

13
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 hardware-accelerated. 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.

Java 2D has built-in 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()` return`Translucency.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

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.

  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 `VolatileImages`. 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 `VolatileImage`, 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 `Image.flush()` 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
- Font 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]
C  [ntdll.dll+0x18238]
C  [ntdll.dll+0x11c76]
C  [MSVCR71.dll+0x16b3]
C  [MSVCR71.dll+0x16db]
C  [fontmanager.dll+0x21f9a]
C  [fontmanager.dll+0x22876]
C  [fontmanager.dll+0x1de40]
C  [fontmanager.dll+0x1da94]
C  [fontmanager.dll+0x48abb]
j  sun.font.FileFont.getGlyphImage(JI)J+0
j  sun.font.FileFontStrike.getGlyphImagePtrs([I[JI)V+92
j  sun.font.GlyphList.mapChars(Lsun/java2d/loops/FontInfo;I)Z+37
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.

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.
- 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 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 the Linux operating system, Swing components perform this adjustment automatically.

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.

Font 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

This section describes some issues that can happen with Java 2D printing and suggests causes and solutions.

See Printing questions in the Java 2D FAQ for additional help.

1. **Java runtime crashes during printing on Windows**.

   Cause: The Java runtime 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 Java runtime 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 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 `lpc` and `lpstat` in the standard location, as previously mentioned.

14
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

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 terminate the X server for Linux.

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

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 API documentation, 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;
                    }
                }
                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 `JComponent` to overlap.

In this case, the parent must override `isOptimizedDrawingEnabled` to return `false`. If you do not override `isOptimizedDrawingEnabled`, 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(false)`

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 (.lnk 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.

15
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.

16
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). 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 Linux operating system, 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 Oracle, 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 `SourceDataLine` for playing, then a separate thread for writing data into the line is also the preferred method to use. If you use `Clip` for playing, then the `Clip` 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.

17

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

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.

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 was already fixed.

> **Note:**
>
> It is always recommended to test with the available latest update release to see if the issue persists.

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:

> **Note:**
>
> First, test with the latest update release to see if the issue persists. Frequently, if a bug report is submitted for an older release, then test with the available latest available update release or even a latest available early access (EA) release. The EA release may contain new features and bug fixes.

- 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.
- 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 bug or a similar bug was reported. If the bug has already been reported, then the bug report might have further information, such as the following:
  - If the bug was already fixed, then the release in which it was fixed is given.
  - A workaround for the problem.
  - Comments in the evaluation that explain, in further detail, the circumstances that cause the bug to happen.
- If you conclude that the bug was not already reported, then submit a new bug.

Before submitting a bug, verify that the environment where the problem happens is a supported configuration. See Supported System Configurations.

In addition to the system configurations, check the list of supported locales. See Supported Locales.

Collect Data for a Bug Report

> **Note:**
>
> In general, it is recommended to test with the latest update release or even a latest available early access (EA) release to see if the issue persists , and then collect as much relevant data as possible when you create a bug report or submit a support call.

The following sections list the commands or recommend a general procedure to obtain the data:

- Hardware Details
- Operating System Details
- Java SE Version
- Command-Line Options
- Environment Variables
- Fatal Error Log
- Core and Crash Dump
- Detailed Description of the Problem

- Logs and Traces
- Results from Troubleshooting Steps

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

Operating systems provide commands that you can use to get the operating system details.

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 appropriate version of the `java` command by verifying that the installation `bin` directory appears in your `PATH` environment variable before other installations.

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 `-mx` option) or any `-XX` options that specify HotSpot-specific options.

One of the features in Java SE is garbage collector ergonomics. On server-class machines, the `java` command launches the HotSpot Server VM and a parallel garbage collector. A machine is considered to be a server machine if it has at least two processors and 2 GB or more of memory.

The `-XX:+PrintCommandLineFlags` option can be used to verify the command-line options. This option prints all command-line flags to the VM. The command-line options can also be obtained for a running VM or core file using the `jmap` utility.

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`

In addition, collect the following operating-system-specific environment variables.

- On the Linux operating system, collect the values of the following environment variables:
  - `LD_LIBRARY_PATH`
  - `LD_PRELOAD`
  - `SHELL`
  - `DISPLAY`
  - `HOSTTYPE`
  - `OSTYPE`
  - `ARCH`
  - `MACHTYPE`
- On Linux, also collect the values of the following environment variables:
  - `LD_ASSUME_KERNEL`
  - `_JAVA_SR_SIGNUM`
- On Windows, collect the values of the following environment variables:
  - `OS`
  - `PROCESSOR_IDENTIFIER`
  - `_ALT_JAVA_HOME_DIR`

Fatal Error Log

The fatal error log is created when a fatal error occurs.

> **Note:**
>
> It is recommended to test with the latest update release to see if the problem persists.

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

Core and crash dumps can be very useful when trying to diagnose a system crash or hung process.

The procedure for generating a dump is described in Collect Core Dumps.

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.

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 diagnosis or run test binaries on the system where the issue occurs.

- If the problem is reproducible, 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 (for example, a commercial or open source library or package), then provide then details about where and how to obtain the library.

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 is the option to enable output from the garbage collector.)

In other cases, the output from the `jstat` command can be used to capture statistical information over the time period leading up to the problem.

In the case of a deadlock or a hung VM (for example, due to a loop), the thread stacks can help diagnose the problem. The thread stacks are obtained by pressing `Control+\` on Linux, and Control+Break on Windows.

In general, provide all relevant logs, traces, and other output in the bug report or during the support call.

Results from Troubleshooting Steps

Report all troubleshooting steps and results that have already occurred.

> **Note:**
>
> Before submitting the bug report, be sure to document any troubleshooting steps that were performed.

For example, if the problem is a crash and the application has native libraries, then you might have already run the application with the `-Xcheck:jni` option to reduce the likelihood that the bug is in the native code. Another case could be a crash that occurs with the HotSpot Server VM (`-server` option). If you have also tested with the HotSpot Client VM (`-client` option) and the problem does not occur, then this is an indication that the bug might be specific to the HotSpot Server VM.

In general, include in the bug report all troubleshooting steps and results that have already occurred. This type of information can often reduce the time that is required to diagnose an issue.

Collect Core Dumps

A core dump or a crash dump is a memory snapshot of a running process.

A core dump can be automatically created by the operating system when a fatal or unhandled error (for example, signal or system exception) occurs. Alternatively, a core dump can be forced by using system-provided 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.

When collecting a core dump, be sure to gather other information about the environment so that the core file can be analyzed (for example, OS version, patch information, and the fatal error log).

Core dumps do not usually contain all the memory pages of the crashed or hung process. With each of the operating systems discussed here, the text (or code) pages of the process are not included in core dumps. But, to be useful, a core dump must consist of pages of heap and stack at as a minimum. Collecting non-truncated core dump files is essential for postmortem analysis of the crash.

The following sections describe scenarios for collecting core dumps.

- Collect Core Dumps on Linux
- Reasons for Not Getting a Core File

- Collect Crash Dumps on Windows

Collect Core Dumps on Linux

On Linux, unhandled signals such as segmentation violation, illegal instruction, and so forth, result in a core dump.

By default, the core dump is created in the current working directory of the process and the name of the core dump file is `core.pid`, where `pid` is the process ID of the crashed Java process.

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.

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:

- `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.

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. The following example shows the output when an unexpected signal occurs.

```
#
# ================================================================
# Unexpected Error
# ----------------------------------------------------------------
# SIGSEGV (0xb) at pc=0x06232e5f, pid=11185, tid=8194
# Do you want to debug the problem?
```

```
To debug, run 'gdb /proc/11185/exe 11185'; then switch to thread 8194
Enter 'yes' to launch gdb automatically (PATH must include gdb)
Otherwise, press RETURN to abort...
================================================================================
```

Enter `yes` to launch the `gdb` (GNU Debugger) interface, as suggested by the error report shown. In the `gdb` prompt, you can give the `gcore` command. This command creates a core dump of the debugged process with the name core.*`pid`*, where *`pid`* is the process ID of the crashed process. Ensure that the `gdb gcore` command is supported in your versions of `gdb`. Look for `help gcore` in the `gdb` command prompt.

Reasons for Not Getting a Core File

The following is a list of reasons what a core file might not be generated on Linux:

- The user does not have permission to write in the current working directory of the process.
- The 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 or there is no space left.
- The current directory has a subdirectory named `core`.
- The current working directory is remote. It might be mapped by a Network File System (NFS), and NFS failed at the time the core dump was about to be created.
- The core file size limit is too low. Check your core file size limit using the `ulimit -c` command (Bash shell) or the `limit -c` command (C shell). If the output from this command is not unlimited, then the core dump file size might not be large enough. If this is the case, then you will get truncated core dumps or no core dump at all. In addition, ensure that any scripts that are used to launch the VM or your application do not disable core dump creation.
- 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 no core dump, 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:**

  On the Windows operating system, 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** checkbox.

  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.

  As with the Linux operating system, 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

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_error`*`pid`*`.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 Linux operating system, the temporary directory is `/tmp`. On 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 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 line is the function frame that caused the crash, as shown in the following example.

Table A-2 Line Description

Line Component	Description
C	Frame type

Table A-2 (Cont.) Line Description

Line Component	Description
`[libMyApp.so+0x57d] Java_MyApp_readData+0x11`	Same as pc, 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
C	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

The thread section of the log contains 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 the Linux operating system, 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=0x0000000000000000, RBX=0x00007f0f17aff3b0,
RCX=0x0000000000000001, RDX=0x00007f1033880358
RSP=0x00007f10346be930, RBP=0x00007f10346be930,
RSI=0x00007f10346be9a0, RDI=0x00007f102c013218
R8 =0x00007f0f17aff3b0, R9 =0x0000000000000008,
R10=0x00007f1011bb1de9, R11=0x0000000101cfc5e0
R12=0x0000000000000000, R13=0x00007f0f17aff3b0,
R14=0x00007f10346be9a8, R15=0x00007f102c013000
```

```
RIP=0x00007f0f159f857d, EFLAGS=0x0000000000010283,
CSGSFS=0x0000000000000033, ERR=0x0000000000000004
```

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 0000000000000000

Instructions: (pc=0x00007f0f159f857d)
0x00007f0f159f855d:   3d e6 08 20 00 ff e0 0f 1f 40 00 5d c3 90 90 55
0x00007f0f159f856d:   48 89 e5 48 89 7d f8 48 89 75 f0 b8 00 00 00 00
0x00007f0f159f857d:   8b 00 5d c3 90 90 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
j  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+0x8a7f1e]  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 and 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)]
```

Table A-6 Thread List Description

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

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.

```
Java Threads: ( => current thread )
  0x00007f102c469800 JavaThread "C2 CompilerThread0" daemon
[_thread_blocked, id=18302, stack(0x00007f0f16f31000,0x00007f0f17032000)]
  0x00007f102c468000 JavaThread "Signal Dispatcher" daemon [_thread_blocked,
id=18301, stack(0x00007f0f17032000,0x00007f0f17133000)]
  0x00007f102c450800 JavaThread "Finalizer" daemon [_thread_blocked,
id=18298, stack(0x00007f0f173fc000,0x00007f0f174fd000)]
  0x00007f102c448800 JavaThread "Reference Handler" daemon [_thread_blocked,
id=18297, stack(0x00007f0f174fd000,0x00007f0f175fe000)]
=>0x00007f102c013000 JavaThread "main" [_thread_in_native, id=18245,
stack(0x00007f10345c0000,0x00007f10346c0000)]

Other Threads:
  0x00007f102c43f000 VMThread "VM Thread" [stack:
0x00007f0f175ff000,0x00007f0f176ff000] [id=18296]
  0x00007f102c54b000 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.

Table A-7 VM 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.

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, 0x46bb0000,
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 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
102000000-800000000 ---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 omitted.
```

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             |
```

Table A-8 Memory Map Format Description

Memory Map Component	Description
`40049000-4035c000`	Memory region
`r-xp`	Permission: • read • write • execute • private • share
`00000000`	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

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.dll
0x77dd0000 - 0x77e5d000     C:\WINDOWS\system32\ADVAPI32.dll
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
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 Linux operating system, the next information in the error log is the list of signal handlers, as shown in the following example.

```
Signal Handlers:
SIGSEGV: [libjvm.so+0xd48840], sa_mask[0]=11111111011111111101111111111110,
sa_flags=SA_RESTART|SA_SIGINFO
SIGBUS: [libjvm.so+0xd48840], sa_mask[0]=11111111011111111101111111111110,
sa_flags=SA_RESTART|SA_SIGINFO
SIGFPE: [libjvm.so+0xd48840], sa_mask[0]=11111111011111111101111111111110,
sa_flags=SA_RESTART|SA_SIGINFO
SIGPIPE: [libjvm.so+0xb60080], sa_mask[0]=11111111011111111101111111111110,
sa_flags=SA_RESTART|SA_SIGINFO
SIGXFSZ: [libjvm.so+0xb60080], sa_mask[0]=11111111011111111101111111111110,
sa_flags=SA_RESTART|SA_SIGINFO
SIGILL: [libjvm.so+0xd48840], sa_mask[0]=11111111011111111101111111111110,
sa_flags=SA_RESTART|SA_SIGINFO
SIGUSR2: [libjvm.so+0xb5ff40], sa_mask[0]=00000000000000000000000000000000,
sa_flags=SA_RESTART|SA_SIGINFO
SIGHUP: [libjvm.so+0xb60150], sa_mask[0]=11111111011111111101111111111110,
sa_flags=SA_RESTART|SA_SIGINFO
SIGINT: [libjvm.so+0xb60150], sa_mask[0]=11111111011111111101111111111110,
sa_flags=SA_RESTART|SA_SIGINFO
```

```
SIGTERM: [libjvm.so+0xb60150],
sa_mask[0]=11111111011111111101111111111110, sa_flags=SA_RESTART|
SA_SIGINFO
SIGQUIT: [libjvm.so+0xb60150],
sa_mask[0]=11111111011111111101111111111110, sa_flags=SA_RESTART|
SA_SIGINFO
```

System Section Format

The final section in the error log is the system information. The output is operating-system-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.

```
---------------  S Y S T E M  ---------------

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
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:         976092 kB
Dirty:                40 kB
Writeback:             0 kB
AnonPages:        912404 kB
Mapped:            95804 kB
Shmem:              2936 kB
Slab:           28625980 kB
SReclaimable:   28337400 kB
SUnreclaim:       288580 kB
KernelStack:        6040 kB
PageTables:        42524 kB
```

```
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:    75324364 kB
Committed_AS:    6172612 kB
VmallocTotal:   34359738367 kB
VmallocUsed:      681668 kB
VmallocChunk:   34282379392 kB
HardwareCorrupted:     0 kB
AnonHugePages:         0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:      171520 kB
DirectMap2M:     8208384 kB
DirectMap1G:    142606336 kB

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 pdpe1gb
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 (9-
ea+167), built on Apr 27 2017 00:28:45 by "javare" with gcc 4.9.2
```

On the 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 the Linux system, the `uname` system call is 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 `rpm -q glibc` might provide more detailed version information.

On the Linux operating system, 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
```

Table A-9 rlimit Description

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`)

```
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 • `4` - i486 • `5` - Pentium • `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.

Table A-14 AMD64/EM64T Features

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 Linux

Table B-1 describes the default values of some useful properties on the Linux platform.

Table B-1 Default Java 2D Properties on Linux

Setup	DGA	SHM	Pixmaps	OnScreen	OffScreen
Linux, SunRay, VNC	Off	On	On	X11/MITSHM	Shared/Server Pixmaps
Remote X server, ssh	Off	Off	On	X11	Server Pixmaps

The following list explains how to change the defaults.

- The X11 pipeline is the default pipeline for 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.

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.

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 prints trace messages during execution.

This option can be useful when diagnosing issues involving the security libraries in the JDK.

To learn more about the `java.security.debug` system property, see 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

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 `java.lang.OutOfMemoryError` 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 ...
```

```
Heap dump file created [531535128 bytes in 14.691 secs]
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at ConsumeHeap$BigObject.(ConsumeHeap.java:22)
        at ConsumeHeap.main(ConsumeHeap.java:32)
```

The `ConsumeHeap` fills the Java heap and runs out of memory. When a `java.lang.OutOfMemoryError` 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 `java_pid`*`pid.hprof`* in the working directory of the VM, as in the example above. You can specify an alternative file name or directory with the `-XX:HeapDumpPath=` option. For example `-XX:HeapDumpPath=/disk2/dumps` will cause the heap dump to be generated in the `/disk2/dumps` directory.

The -XX:OnError Option

When a fatal error occurs, the Java HotSpot VM can optionally execute a user-supplied 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="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="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 Linux operating system, 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=0x2000000001164db1, 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`, then`gdb` will be launched (assuming it is set in the `PATH` variable).

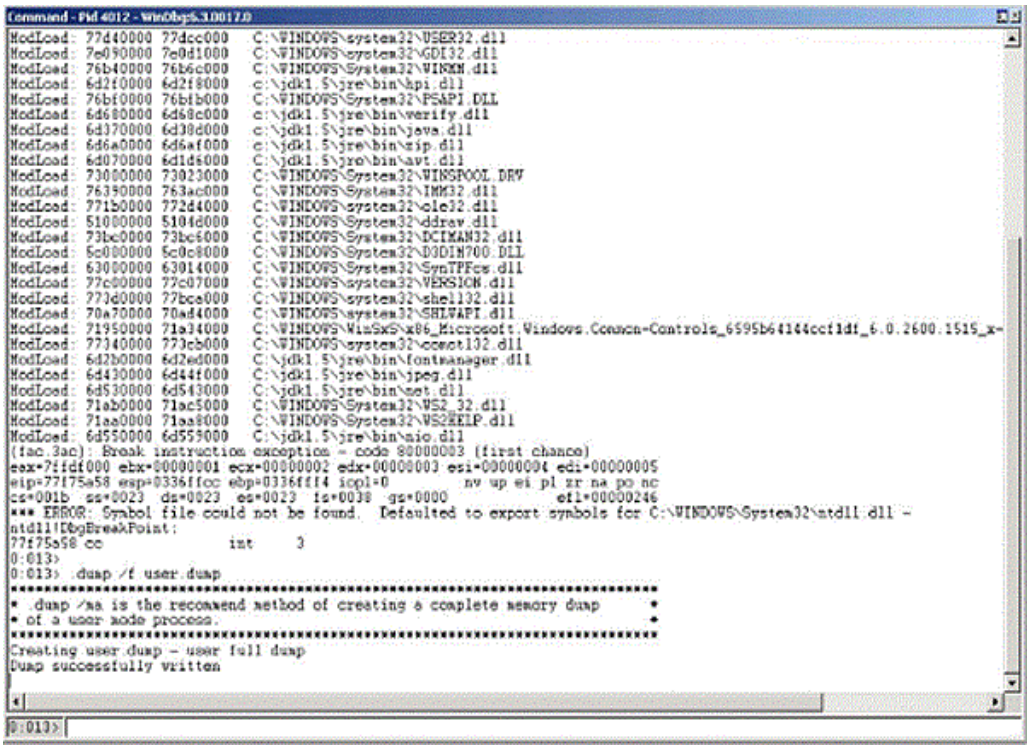
On Windows a message box is displayed. If the user clicks **Yes**, the VM will attempt to start the default debugger. This debugger is configured by a registry setting which is described in Collect Crash Dumps on Windows. If Microsoft Visual Studio is installed, the default debugger is typically configured 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 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



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 a `java.lang.OutOfMemoryError` 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 The java.lang.OutOfMemoryError Error.

- **`-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:+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.

```
FATAL ERROR in native method: Null object passed to JNI
    at java.net.PlainSocketImpl.socketAccept(Native Method)
    at java.net.PlainSocketImpl.accept(PlainSocketImpl.java:343)
    - locked <0x450b9f70> (a java.net.PlainSocketImpl)
    at java.net.ServerSocket.implAccept(ServerSocket.java:439)
    at java.net.ServerSocket.accept(ServerSocket.java:410)
    at org.apache.tomcat.service.PoolTcpEndpoint.acceptSocket
                        (PoolTcpEndpoint.java:286)
    at org.apache.tomcat.service.TcpWorkerThread.runIt
                        (PoolTcpEndpoint.java:402)
    at org.apache.tomcat.util.ThreadPool$ControlRunnable.run
                        (ThreadPool.java:498)
    at java.lang.Thread.run(Thread.java:536)
```

The following example shows an incorrect argument that was provided to a JNI function that expects a `jfieldID` argument.

```
FATAL ERROR in native method: Instance field not found in JNI get/set
                        field operations
        at java.net.PlainSocketImpl.socketBind(Native Method)
        at java.net.PlainSocketImpl.bind(PlainSocketImpl.java:359)
        - locked <0xf082f290> (a java.net.PlainSocketImpl)
        at java.net.ServerSocket.bind(ServerSocket.java:318)
        at java.net.ServerSocket.<init>(ServerSocket.java:185)
        at jvm003a.<init>(jvm003.java:190)
        at jvm003a.<init>(jvm003.java:151)
        at jvm003.run(jvm003.java:51)
        at jvm003.main(jvm003.java:30)
```

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.

More information about these options, along with detailed information about GC analysis and tuning are described in the GC Portal article.

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 provides a summary of tools available in the current release of the JDK.

All of the JDK troubleshooting tools that are described in this document are available on Linux.

The following JDK troubleshooting tools are also available on Windows:

- Flight Recorder
- `jcmd`
- JConsole
- Java Virtual Machine
- `jdb`
- `jinfo`
- `jmap`
- `jps`
- `jrunscript`
- `jstack`
- `jstat`
- `jstatd`
- `visualgc`