# Troubleshooting Guide for Java SE 6 Desktop Technologies

Part No: 821–1866
September 2008

ORACLE®

100419@23626

# Contents

# Preface

This document helps in troubleshooting problems that might occur with applications that use the desktop technologies in the release of Java™ Platform, Standard Edition Development Kit 6 (JDK™ 6 release or Java SE 6 release).

Most of the information in this guide also applies to the Java SE 5 release (also known as Java 2 SE 1.5 or 5.0). Information that applies only to Java SE 6 is indicated accordingly.

For help in troubleshooting possible problems between the application and the Java HotSpot™ virtual machine, see the *Troubleshooting Guide for Java SE 6 with HotSpot VM* or the *Java 2 SE 5.0 Troubleshooting and Diagnostic Guide*.

## Who Should Use This Guide

The target audience for this document comprises developers who are working with the desktop technologies in Java SE 5 or 6, as well as support or administration personnel who maintain applications that are deployed with Java SE 5 or 6.

This document is intended for readers with a detailed understanding of the desktop technologies, a high-level understanding of the components of the Java Virtual Machine, as well as some understanding of concepts such as garbage collection, threads, native libraries, and so on. In addition, it is assumed that the reader is reasonably proficient on the operating system where the Java SE application is installed.

## How This Guide Is Organized

The first chapter of this guide introduces the desktop technologies, presents some general troubleshooting information and guidelines, and introduces some troubleshooting tools.

Further chapters suggest procedures to try when you encounter problems with some of the desktop technologies, namely, AWT, Java 2D, Swing, Internationalization, Java Sound, and Java Plug-in. More desktop technologies will be gradually added to this guide.

The last chapter provides suggestions on what to try before submitting a bug report, guidance on how to submit a report, and suggestions on what data to collect for the report.

Finally, there is an appendix for each of the following reference areas: Java 2D properties, and details about the format of the fatal error report.

## Feedback and Suggestions

Troubleshooting is a very important topic. If you have feedback on this document or if you have suggestions for topics that could be covered in a future version, use the Feedback Form (`http://developers.sun.com/contact/feedback.jsp?category=javase`). Fill in the relevant fields and click Send.

---

**Note –** Do not use this feedback form for support requests; they will not be answered. Technical support is provided at the Services site for Sun Developer Network (`http://developers.sun.com/services`).

---

## Other Resources

The following additional online troubleshooting resources are available:

- Java SE Desktop Overview site (`http://java.sun.com/javase/technologies/desktop/index.jsp`)
- Java Technologies Home site (`http://java.sun.com/products/`)

## Commercial Support

Sun provides a wide range of support offerings, from developer technical support for software developers using Sun development products or technologies, to support for production systems in enterprise environments. Two commercial support options are summarized here: developer technical support and Java mulitplatform support.

### Developer Technical Support

Developer technical support is aimed at developers who are using Sun development products or technologies, and who are working at the source-code level of their own applications.

This support offering includes response to technical questions, diagnostic and troubleshooting help, suggestions for best practices, bug escalation, and more.

Details on these support offerings are provided at the Services site for Sun Developer Network (`http://developers.sun.com/services`). This site steers you to information about developer support from Sun, for example:

- Sun Developer Solutions site (`http://developers.sun.com/`)
- Sun Solution Support Engineering Services (`http://www.sun.com/service/sse/`)

## Java Multiplatform Support

The Java multiplatform support offering is designed to provide production support for shipping releases of Java technology-based applications using Sun's Java runtime environment (JRE) and distributed to end users in heterogeneous environments. This support offering helps to optimize application performance and to reduce time spent keeping applications up and running.

The highest level of this support offering can include accelerated access to an engineer and emergency software fixes. Details on this support offering are available at the SunSpectrum Java MultiPlatform site (`http://www.sun.com/service/javamultiplatform/index.xml`).

## Community Support

Community support can often be obtained using the Java Technology Forums. The forums provide a way to share information and locate solutions to problems. The forums are located at `http://forums.java.sun.com`.

## Typographic Conventions

The following table describes the typographic conventions that are used in this book.

**TABLE P–1** Typographic Conventions

| Typeface | Meaning | Example |
|---|---|---|
| `AaBbCc123` | The names of commands, files, and directories, and onscreen computer output | Edit your `.login` file.<br>Use `ls -a` to list all files.<br>`machine_name% you have mail.` |
| **`AaBbCc123`** | What you type, contrasted with onscreen computer output | `machine_name%` **`su`**<br>`Password:` |
| *aabbcc123* | Placeholder: replace with a real name or value | The command to remove a file is `rm` *filename*. |

**TABLE P–1** Typographic Conventions *(Continued)*

| Typeface | Meaning | Example |
|---|---|---|
| *AaBbCc123* | Book titles, new terms, and terms to be emphasized | Read Chapter 6 in the *User's Guide*.<br>A *cache* is a copy that is stored locally.<br>Do *not* save the file.<br>**Note:** Some emphasized items appear bold online. |

## Shell Prompts in Command Examples

The following table shows the default UNIX® system prompt and superuser prompt for shells that are included in the Solaris OS. Note that the default system prompt that is displayed in command examples varies, depending on the Solaris release.

**TABLE P–2** Shell Prompts

| Shell | Prompt |
|---|---|
| Bash shell, Korn shell, and Bourne shell | `$` |
| Bash shell, Korn shell, and Bourne shell for superuser | `#` |
| C shell | `machine_name%` |
| C shell for superuser | `machine_name#` |

## Related Third-Party Web Site References

Third-party URLs are referenced in this document and provide additional, related information.

**Note –** Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

## Acknowledgments

Many people contributed input to this guide: Artem Ananiev, Kannan Balasubramanian, Tim Bell, Christopher Campbell, Dmitry Cherepanov, Uday Dhanikonda (responsible engineering manager), Andrei Dmitriev, Denis S. Fokin, Alexander Gerasimov, Jennifer Godinez, Jim Holmlund, Yuka Kamiya, Antonia Lewis (writer), Alexy Menkov, Igor Nekrestyanov, Yuri Nesterenko, Phil Race, Oleg Semenov, Oleg Sukhodolsky, Anton Tarasov, Dmitri Trembovetski (major contributor and coordinator of input and review from Java Client Group).

## Document History

The following table tracks the changes in versions of this guide.

| Version Date | Changes |
|---|---|
| December 2006 | Original version |
| January 2007 | New feedback form; several minor corrections, reorganization, reformatting. |
| August 2007 | Document now applies to both releases 5 and 6 of Java SE. Added advice about when to debug Swing remotely. Format improvements. |
| September 2008 | Fixed broken links. |

CHAPTER 1

# Introduction

This chapter explains how the different Java SE client technologies interact with each other. In addition, the chapter helps you to pinpoint the technology where you might start troubleshooting your problem.

## 1.1 Overview of Java SE Desktop Technologies

Java SE Desktop consists of several technologies, as described at the Java SE Desktop Overview site (`http://java.sun.com/javase/technologies/desktop/index.jsp`). This guide currently describes troubleshooting procedures for the following desktop technologies:

- AWT
- Java 2D
- Swing
- Internationalization
- Java Sound
- Java Plug-in

This section explains how the different desktop technologies interact with each other. It also describes which of the technologies are more likely to work with native code, or depend on the environment, for example, the hardware or software configuration.

The better you understand the relationships among these technologies, the more quickly you can pinpoint the area your problem falls into.

### 1.1.1 Introduction to Java SE Desktop

The Java SE Desktop Java technologies are used to create rich client applications and applets. The desktop tools and libraries interface with the core tools and libraries of the platform.

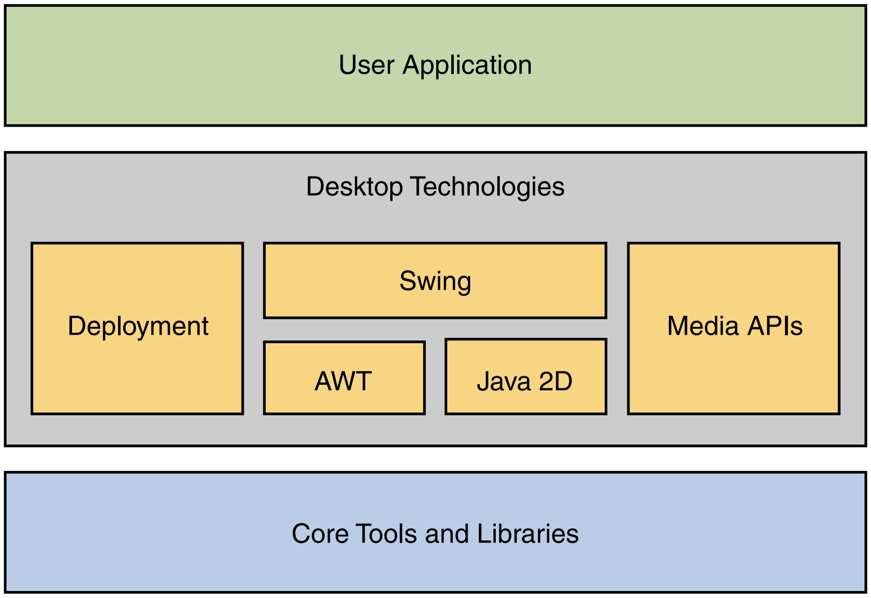


FIGURE 1–1 Overview of Java SE Desktop

### 1.1.2 AWT

The Abstract Window Toolkit (AWT) supports Graphical User Interface (GUI) programming. This API gives you, the developer, the following capabilities:

- Construction of components such as menus, buttons, text fields, dialog boxes, and checkboxes
- Handling of user input through those components
- Rendering of simple shapes such as ovals and polygons
- Control over the user-interface layout and fonts used by your applications
- Handling of native drag-and-drop, events, and cursors

These classes 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.

### 1.1.3 Java 2D

The Java 2D API is a set of classes for advanced 2D graphics and imaging. This API enhances the graphics, text, and imaging capabilities of the AWT.

Java 2D is also at the bottom of the software stack (closest to the underlying operating and desktop system).

### 1.1.4 Swing

The Swing classes are built on top of the AWT and Java 2D architecture. Swing implements a set of components for building GUIs and adding rich graphics functionality and interaction to Java applications, with a pluggable look and feel.

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.

### 1.1.5 Internationalization

Internationalization is the process of designing software so that it can be adapted to various languages and regions without engineering changes.

### 1.1.6 Sound

Java Sound provides a low-level API to control the input and output of sound operations, including capturing, processing, and playing back audio and MIDI (Musical Instrument Digital Interface) data.

### 1.1.7 Plug-in

Java Plug-in extends the functionality of a web browser by allowing applets or Java Beans to be run under the Java SE runtime environment (JRE) rather than the Java runtime environment that is delivered with the web browser.

## 1.2 General Troubleshooting Tips

The following list provides some advice that can help you in troubleshooting a problem.

1. Identify the symptom.
   - Identify the type of issue.
   - Find the problem area.
   - Note relevant configuration information.
2. Eliminate non-issues.
   - Make sure 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 single and 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. Several unsupported properties are described on the System Properties for Java 2D Technology page (`http://java.sun.com/javase/6/docs/technotes/guides/2d/flags.html`).
4. Find the fix.
   - Find a possible workaround.
   - Fix the setup.
   - File a bug.
   - Fix the user's application.

The *Troubleshooting Guide for Java SE 6 with HotSpot VM* provides information that can help in troubleshooting problems between Java SE applications and the Java HotSpot virtual machine. A lot of this information can also be applied to problems with applications that use the Java SE desktop technologies.

See Chapter 8, “Submitting Bug Reports,” for guidance on how to submit a bug report and suggestions on what data to collect for the report.

## 1.3 Identifying the Problem Area

Take a moment to categorize the problem you are experiencing. This will help you identify the specific area of the problem, find the cause, and ultimately determine a solution or a workaround. Some of these issues might seem obvious, but it is always helpful to consider every possibility and to eliminate what is not an issue.

### 1.3.1 Crashes

When a crash occurs, an error log is created with information and the state obtained at the time of the fatal error. The default name of the error log file is `hs_err_pid`*pid*`.log`. For a standalone Java application this file is created in the current directory, while for Java applets it is created in the browser binaries directory or user client folder.

See Appendix B, “Fatal Error Log,” for a detailed description of the fatal error log.

A line near the top of the header section indicates the library where the error occurred. The example below 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 JNI native code, 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 mostly concerned with native look and feel, and if your application is using native look and feel, 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 JDK usually indicate problems with the environment, for example, bad video drivers or desktop managers.

In the case of a VM crash, see the *Troubleshooting Guide for Java SE 6 with HotSpot VM*.

### 1.3.2 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.

The following are a few examples of performance-related problems.

- 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 TaskManager on Windows or `top` and `prstat` on Solaris OS and Linux.
- Runtime performance. 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, does the animation look smooth?

  Note that 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 the application does not appear to be slow?

## 1.3.3 Behavior Problems

In addition to crashes, various behavior-related problems can occur. This section presents some of these problems and attempts to guide you to the Java SE desktop technology to troubleshoot.

- Hangs: The application does not respond to user input. See the chapter on hanging or looping processes in the *Troubleshooting Guide for Java SE 6 with HotSpot VM*.
- Exceptions in Java code. Exceptions are visibly thrown in the console or the application log files. An examination of this output will guide you to the problem area.
- Rendering and repainting issues. For example, the application has an incorrect appearance after a repaint that was caused by another application being dragged over. Other examples are incorrect font, wrong colors, scrolling, damaging the application's frame by dragging another window over it, and updating a damaged area. These issues indicate a problem in Java 2D, or perhaps in Swing.

  If your application exhibits repainting issues, the problem is likely to be with Java 2D or Swing.

  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 Solaris OS or Linux), it is very likely to be a Swing `PaintManager` problem.

  See Chapter 3, "Troubleshooting Java 2D," for ways of changing Java 2D rendering pipelines with some flags. This can also help in determining 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 interaction with a non-default screen device).
- Desktop-interaction-related issues indicate a problem in AWT.

Some examples of desktop-interaction-related issues are resizing the window, minimizing or maximizing them, focus handling problems, modality, system Tray or Splashscreens, window placement on the screen, handling (enumerating) of multiple screens.

- Drag-and-drop problems. Drag-and-drop belongs to AWT.
- Printing problems. Depending on the API that is used, this could be either in Java 2D or AWT.
- Font problems.

  Font rendering issues in AWT applications might be a problem in font properties or in internationalization.

  However, if your application is purely AWT, the text rendering problems might be caused by Java 2D. On Solaris OS or Linux the text rendering is performed by Java 2D. Alternatively, the problem might be in AWT or in internationalization.
- Painting problems. This is most likely a Swing issue.
- Text-rendering quality.

  The 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 the font rendering), the problem is likely to be in Java 2D.
- Full-screen issues. This is a Java 2D API.
- Encoding and locales issues (for example, no locale-specific characters displayed) indicate internalization problems.

## 1.4 Basic Tools

This section simply lists a few tools that can help in troubleshooting. The *Troubleshooting Guide for Java SE 6 with HotSpot VM* contains detailed information on most of these tools, as well as many other useful tools.

- Performance: benchmarks, profilers, DTrace, Java probes
- Footprint: `jhat`, `jmap`, profilers
- Crashes: native debuggers
- Hangs: JConsole, `jstack`, ctrl-break
- Check for bad fonts: Font2DTest (delivered with the JDK in `demo/jfc/Font2DTest`)

You can also debug JDK builds from `dev.java.net` for various issues.

## 1.5 Using JDWP for Debugging

JDWP (Java Debugging Wire Protocol) is very useful in debugging applications as well as applets.

Perform the following steps to debug applications:

1. Open a command line window. Set the `PATH` environment variable to *jdk*`/bin`.
2. Run the Java program (called `Test` in this example) to be debugged as follows:
   - On Windows:
     ```
     java -Xdebug -Xrunjdwp:transport=dt_shmem,address=debug,server=y,suspend=y Test
     ```
   - On Solaris OS and Linux:
     ```
     java -Xdebug -Xrunjdwp:transport=dt_socket,address=8888,server=y,suspend=y Test
     ```
3. The `Test` class starts in a debugging mode and waits for a debugger to attach to it.
4. To do Java level debugging, open another command line window and run `jdb` to attach to the above running debug server at address `debug` (Windows) or `8888` (Solaris OS or Linux).

   **`jdb -attach 'debug'`** *or* **`jdb -attach 8888`**
5. After `jdb` initializes and attaches to `Test`, set your breakpoints and run.
   ```
   stop in Test.main
   run
   ```
6. The `jdb` utility will hit the breakpoint.

To perform native level debugging along with Java debugging, use native debuggers to attach to the Java process running with JDWP.

In Windows, perform the following steps:

1. Open Visual Studio.
2. Choose Build → Start Debug → Attach to Process. Select the Java process that is running with JDWP.
3. Choose Project → Settings → Additional DLLs. Add the native `dll` that you want to debug, for example `Test.dll`.
4. Open the source file (one or more) of `Test.dll` and set your breakpoints.
5. Type `cont` in the `jdb` window. The process will hit the breakpoint in Visual Studio.

On Solaris OS, you can use the `dbx` utility to do native level debugging, and on Linux you can use the `gdb` utility.

Perform the following steps to debug applets using JDWP:

1. Launch the Java Control Panel.
2. Set the Applet Runtime settings.

3. In the field Java Runtime Parameters, enter the following:
   - On Windows:

     ```
     Djavaplugin.trace=true -Xdebug -Xrunjdwp:
     transport=dt_shmem,address=debug,server=y,suspend=y
     ```
   - On Solaris OS and Linux:

     ```
     Djavaplugin.trace=true -Xdebug -Xrunjdwp:
     transport=dt_shmem,address=debug,server=y,suspend=y
     ```
4. When you launch a browser and load an applet, the Java plugin starts in debugging mode and waits for a debugger to attach to it at the address `debug` (Windows) or `8888` (Solaris OS or Linux).
5. Run `jdb` from a command window and attach to address `debug` (or `8888`).

   ```
   jdb -attach debug
   Initializing jdb ...
   VM Started: No frames on the current call stack
   main[1]
   ```
6. After `jdb` initializes, set the breakpoints, and then run.

   ```
   sStop in MyApplet.func1
   run
   ```
7. The applet will run in the browser until it hits the set breakpoint. Then you can debug the applet, see the control flow, watch its variables, and so forth.

◆ ◆ ◆ CHAPTER 2

# Troubleshooting AWT

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 AWT API:

## 2.1 Debugging Tips for AWT

The following AWT debugging tips can be helpful:

- Solaris OS and Linux only: To trace X11 errors, set `NOISY_AWT=true`.
- To dump the AWT component hierarchy, press Ctrl+Shift+F1.
- If the application hangs, get a stack trace with Ctrl+\ (SIGQUIT) on Solaris OS and Linux or Ctrl+Break on Windows.

## 2.2 Problems With Layout

This section describes some possible problems with layout and provides workarounds when available.

**Issue: Call to `invalidate()` and `validate()` increases Component size.**

*Cause:* Due to some specifics of the layout manager `GridBagLayout`, if `ipadx` or `ipady` is set, and if `invalidate()` and `validate()` are called, then Component size increases to the value of ipadx or ipady. This happens because the layout manager GridBagLayout 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 very simple workaround. Use components with the overridden method getPreferredSize(), which always returns the current needed size.

```
public Dimension getPreferredSize(){
   return new Dimension(size+xpad*2+1, size+ypad*2+1);
}
```

**Issue: 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().

## 2.3 Key Events

This section describes issues with key events.

### 2.3.1 General Unresolved Keyboard Issues

The following keyboard issues are currently unresolved.

- On some non-English keyboards certain accented keys are engraved on the keytop 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 charCode for the WM_CHAR message. AWT should distinguish them. This will be fixed in a future release.

### 2.3.2 Linux and Solaris 10 OS x86 Keyboard Issues

The following keyboard issues concern the Linux and Solaris 10 OS x86 systems.

- Keyboard input in these systems is usually based on XKEYBOARD X Window extension. Users can configure only one keyboard layout (for instance, Danish: dk) or several layouts to switch between (for example, us and dk).
- With some keyboard layouts, for instance sk, hu, and cz, pressing the NumPad decimal separator not only enters a delimiter but also deletes the previous character. This is due to a native bug. A workaround is to use two layouts, for example, us and sk. In this case the numeric keypad works correctly in both layouts.

- On UNIX systems that support dynamic keyboard changes, a running Java application does not recognize such a change. For instance, changing the keyboard from US to German does not change the keyboard mapping. Although the X server detects the change and sends out a `MappingNotify` event to interested clients, AWT does not refresh its notion of the keycode-keysym mapping.

## 2.4 Modality

With the Java SE 6 release, many problems were fixed and many improvements were implemented in the area of AWT modality. If you observe a modality problem with Java SE 1.5 or an earlier release, first upgrade to the Java SE 6 release to see if the problem has been already fixed. Some of the problems that were fixed in Java SE 6 are the following:

- Modal dialog goes behind a blocked frame.
- Two modal dialogs with the same parent window opened at the same time.

### 2.4.1 UNIX Window Managers

Many of the modality improvements are unavailable in some Solaris OS or Linux environments, for example, when using CDE window managers. With Java SE 6, to see if a modality type or modal exclusion type is supported in particular configuration, use the methods `Toolkit.isModalityTypeSupported()` and `Toolkit.isModalExclusionTypeSupported()`.

Another problem exists when running Java modal dialogs on Solaris OS or Linux. When a modal dialog appears on the screen, the window manager might hide some of the Java top-level windows in the same application from the task bar. 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.

### 2.4.2 Using Modal Dialogs from Applets

When your application runs as an applet in a browser and shows a modal dialog, the browser window might become blocked. The implementation of this blocking varies in different browsers and operating systems. For example, on Windows, both Internet Explorer and Mozilla work correctly, and on Solaris OS and Linux, Mozilla windows are not blocked. This will be corrected in a future release.

### 2.4.3 Other Modal Problems

The The AWT Modality document for Java SE 6 (`https://java.sun.com/javase/6/docs/api/java/awt/doc-files/Modality.html`) describes the modality-related features and how to use them. One of the sections in this document describes some areas that might be related to or affected by modal dialogs: always-on-top property, focus handling, window states, and so forth. Application behavior in such cases is usually unspecified or depends on the platform; therefore, do not rely on any particular behavior.

## 2.5 Memory Leaks

This section first describes how to troubleshoot memory leaks. It then presents some possible sources of memory leaks and provides workarounds.

### 2.5.1 Troubleshooting Memory Leaks

To get more information on a memory leak, execute `java` with the heap profiler active. Specify that the output should be generated in binary format so that you can use the `jhat` utility to read the output.

```
$ java -agentlib:hprof=file=snapshot.hprof,format=b application
```

See the *Troubleshooting Guide for Java SE 6 with HotSpot VM* for more detailed information on troubleshooting memory leaks, as well as descriptions of the `jhat` utility and other troubleshooting tools that are available.

### 2.5.2 Memory Leak Issues

**Issue: Memory leak in application.**

*Cause:* Frames and Dialogs are sometimes not being garbage-collected. This bug will be corrected in a future version of Java SE.

*Workaround:* Known memory leaks occur in cases when the system starts to transfer focus to a focusable top-level element (window, dialog, frame), but the element is closed, hidden, or disposed of before the focus transfer is complete. Therefore, the application must wait for the focus transfer operation to finish before closing, hiding, or disposing of the element.

Note that this problem normally occurs only when these actions are performed programmatically, since the user typically cannot physically perform these actions fast enough to cause the problem.

## 2.6 Crashes

This section describes how to determine if a crash is related to AWT, as well as how to troubleshoot such crashes.

### 2.6.1 How to Distinguish an AWT Crash

When a crash occurs, an error log is created with information and the state obtained at the time of the fatal error. See Appendix B, "Fatal Error Log," for detailed information about this log file.

A line near the top of the file indicates the library where the error occurred. The example below 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]
...
```

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`. Below is an 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.

### 2.6.2 How to Troubleshoot a Crash in AWT

Most of the AWT crashes occur on the Windows platform and are caused by thread races. Many of these problems were fixed in Java SE version 6, so if your crash occurred in an earlier release, first try to determine if the problem is already fixed in the latest release.

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 an 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.

## 2.7 Problems With Focus

This section includes the following information:

- How to trace focus events
- Description of the focus system in the plugin
- Focus models supported by X Window managers
- Focus traversal
- Miscellaneous problems that can occur with focus

### 2.7.1 How to Trace Focus Events

To troubleshoot a problem with the focus, you can trace focus events. Start with just adding a focus listener to the toolkit, as shown here.

```
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.

Remember that 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 that events of losing focus or activation should come first.

#### 2.7.1.1 Communication With Native Focus System

Sometimes a problem may be caused by the native platform. To check this, investigate the native events that are related to focus. Make sure that the window you want to be focused gets activated and 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, in some cases a component that is the focus owner might not correspond to the component on which the native system set focus. This occurs when the focus is set by synthesizing a focus event. Nevertheless, such a focus owner is painted like a normally focused component. A lightweight component is focused by synthesizing a focus event, whereas its nearest heavyweight component is focused in the normal way. A heavyweight component can also have "synthetic focus," for example, the focus owner in an owned window (not a frame or a dialog). In this case native focus is set on a special component, called a "focus proxy." inside the owner. A focus proxy component is not visible to the user and is used only at the native level. This mechanism allows AWT to separate active and focused windows. However, with the focus proxy, tracking the flow of native focus events can become complicated.

On Solaris OS and Linux, 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 that X server and some window managers may nevertheless send focus events to a window. However all such 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 itself does not get any native focus event. On the Solaris OS and Linux platforms, as well as on the Windows platform, AWT uses the focus proxy mechanism. Therefore, focus on the component is set by synthesizing a focus event, whereas the invisible focus proxy has native focus.

A native window that is mapped to a `Window` object (not a `Frame` or `Dialog` object) has the `override-redirect` flag set. Thus the window manager does not notify the window about 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. Since the major processing of focus occurs at the Java level, debugging focus with XToolkit is simpler than with WToolkit.

## 2.7.2 Focus System in the Plugin

An applet is embedded in a browser as a child (though not a direct child) of an `EmbeddedFrame`. This is a special `Frame` that has the ability to communicate with the plugin. From the applet's perspective the `EmbeddedFrame` is a full top-level `Frame`. Managing focus for an `EmbeddedFrame` requires special additional actions. When an applet first starts, the `EmbeddedFrame` does not get activated by default by the native system. The activation is performed by the plugin that triggers a special API provided by the `EmbeddedFrame`. When focus leaves the applet, the `EmbeddedFrame` is also deactivated in a synthesized manner.

It could happen that a lightweight component does not gain focus but the nearest heavyweight does instead. This is due to the native system setting focus on it by itself, bypassing AWT. To work around this problem, set a focus listener to the heavyweight and retransfer focus on the lightweight when necessary.

## 2.7.3 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.

The `focus-follows-mouse` mode is not detected in XAWT in Java SE 6, and this causes problems for simple windows (objects of `java.awt.Windowclass`). Such windows have the override-redirect property, which means that they can be focused only when the mouse button is pressed, and not by hovering over the window. As a workaround, set `MouseListener` on the window and request focus on it when mouse crosses the window borders.

### 2.7.4 Focus Traversal

Sometimes a problem occurs when some object is not traversable. For example, `LayoutFocusTraversalPolicy`, which is the Swing focus traversal policy (FTP), implicitly transfers focus down-cycle. The focus specification says the following:

> During normal forward focus traversal, the Component traversed after a focus cycle root will be the focus-cycle-root's default Component to focus.

However when you transfer focus from some component to a focus cycle root container (for example, a container inside another focus cycle root), the container itself will not get focus and focus will not be transferred down-cycle into the container.

This behavior will be changed in a future release. However, in the Java SE 6 release you can easily change this behavior. First, make the focus cycle root container traversable itself, for example by calling `setFocusable(true)` on the container. Then by default it will become the default component to focus (because it would be the first component in its cycle). You have to set the default component in the cycle implicitly, as follows:

```
theContainer.setFocusTraversalPolicy(new LayoutFocusTraversalPolicy() {
   public Component getDefaultComponent(Container root) {
      return theComponent;
   }
});
```

Now you have your focus cycle traversable. However, because you made the container focusable, it gains focus first when you move to the cycle, and it looks like focus has disappeared. To avoid this behavior you can add `FocusListener` to your container, as follows:

```
theContainer.addFocusListener(new FocusAdapter() {
   public void focusGained(FocusEvent e) {
      theContainer.getFocusTraversalPolicy().
         getDefaultComponent(theContainer).requestFocusInWindow();
   }
});
```

In this way focus will be transferred to the default component in the container's cycle.

### 2.7.5 Miscellaneous Problems With Focus

This section describes some issues that can arise with focus in AWT and suggests solutions.

**Issue: Linux + KDE, XToolkit. Focus cannot be switched between two frames when frame's title is clicked.**

Clicking a component inside a frame causes focus to change.

*Solution:* Check the version of your window manager and upgrade it to 3.0 or greater.

**Issue: You want to manage focus using `KeyListener` to transfer focus in response to Tab/Shift+Tab, but no key events appear.**

*Solution:* To catch traversal key events, you must enable them by calling `Component.setFocusTraversalKeysEnabled(boolean)`.

**Issue: A window is set 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.

**Issue: CDE + dtwm, XToolkit. A window is set unfocusable with a call to `Window.setFocusableWindowState(false)`, but steals focus from the focussed window when it is displayed.**

The window does not receive `WINDOW_GAINED_FOCUS`, but the currently focused window receives `WINDOW_LOST_FOCUS`. Moreover, the window gains native focus (that is, its title is highlighted), but neither the window nor its components can get key events.

*Solution:* This behavior is specific to `dtwm`. The only way to avoid this problem is to switch to JDS.

**Issue: MS Windows. A component requests focus and at the same time is 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 requesting focus and removal if you need to perform these actions on different threads.

**Issue: MS Windows. During the removal of a container containing a component that is the focus owner, the focus remains on some of the removed components inside the container.**

*Solution:* In order not to lose focus completely, you can request it on, for example, the default component of the current focus cycle root (the parent of the container) after the container is removed.

**Issue: 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.

**Issue: MS Windows. When a window is set `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 front (`Frame.toFront()` method) before setting the window to `alwaysOnTop`.

**Issue: When a SplashScreen is shown and a frame is shown after the SplashScreen window closes, the frame does not get activated.**

*Solution:* Bring the frame to front (`Frame.toFront()` method) after showing it (`Frame.setVisible(true)` method).

**Issue: The `WindowFocusListener.windowGainedFocus(WindowEvent)` method does not return the frame's most recent 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 that this approach does not work for the window, only for the frame or dialog.

**Issue: An applet steals focus when it starts.**

*Solution:* This behavior is the default since JDK 1.3. However you might need to prevent the applet from getting focus on startup, for example, if your applet is invisible and does not require focus at all. In this case, you can set to `false` the special parameter `initial_focus` in the HTML tag, as follows:

```
<applet code="MyApplet" width=50 height=50>
<param name=initial_focus value="false">
</applet>
```

**Issue: A window is disabled with `Component.setEnabled(false)`, but does not get totally 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.

## 2.8 Drag and Drop

This section discusses possible problems with Drag and Drop and the clipboard.

### 2.8.1 Debugging Drag and Drop Applications

It is difficult to use a debugger to troubleshoot Drag and Drop, because during the drag–and–drop operation all input is grabbed. Therefore, if you place a breakpoint during drag–and–drop, 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 Drag and Drop:

- Printing all `DataFlavor` instances
- Printing received data

An alternative to remote debugging is the `System.err.println()` function, which prints output without delay.

### 2.8.2 Frequent Issues With Drag and Drop

This section describes some issues that frequently arise with Drag and Drop in AWT and suggests troubleshooting solutions.

**Problem: Pasting a huge 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 flavors (for example, some text and image flavors), 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 flavors are needed, use one of the following `Clipboard` methods instead of `getContents()`:

- `DataFlavor[] getAvailableDataFlavors()`
- `boolean isDataFlavorAvailable(DataFlavorflavor)`
- `Object getData(DataFlavorflavor)`

**Problem: 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, 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 is generated only once.

**Problem: 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 `DataFlavor.javaFileListFlavor` data flavor. 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 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 a code example, see the Work Around section of this bug report:

http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4899516

**Problem: 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 of the following RFE:

http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4874070

**Problem: There is no way to transfer an array using Drag and Drop.**

The `DataFlavor` class has no constructor which handles arrays. The mime type for array contains characters which should be escaped. For example, the following code 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 code, where the quotation marks are escaped:

```
new DataFlavor(DataFlavor.javaJVMLocalObjectMimeType +
"; class=" +
"\"" +
(new String[0]).getClass().getName() +
"\"")
```

For more information, see the following bug report:

http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4276926

**Problem: There are problems using AWT Drag and Drop support with Swing components.**
Various problems can arise, 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 drag–and–drop. Refer to the following documentation:

- Swing Tutorial (`http://java.sun.com/docs/books/tutorial/uiswing/misc/dnd.html`)
- Swing guide (`http://java.sun.com/javase/6/docs/technotes/guides/swing/`)

**Problem: 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 need to have references to the source and target components in the same area of code, but this is not currently implemented in the Drag and Drop 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 drag source listener, as follows:

```
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()`.

**Problem: Transferring of 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, obtain file content when you create transferable, so that `Transferable.getTransferData()` will not be too long.

## 2.9 Other Issues

This section describes other issues in troubleshooting AWT.

### 2.9.1 Splash Screen Issues

This section describes some issues that can arise with the splash screen in AWT with the Java SE 6 release, and suggests solutions.

**Issue: The user specified a jar file with an appropriate `MANIFEST.MF` in `-classpath`, but the splash screen does not work.**
*Solution:* See next solution.

**Issue: 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 jar 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.

**Issue: Translucent png splash screens do not work on Solaris OS and Linux.**
*Solution:* This is a native limitation of X11. On Solaris OS and Linux, the alpha channel of a translucent image will be compared with 50% threshold. Alpha values above 0.5 will make opaque pixels and pixels with alpha below 0.5 completely transparent. Translucency support might improve in future versions of Java SE.

### 2.9.2 Tray Icon Issues

With the Java SE 6 release on the Windows 98 platform, the method `TrayIcon.displayMessage()` is not supported because the native service to display a balloon is not supported on Windows 98.

If a `SecurityManager` is installed, the value of `AWTPermission` must be set to `accessSystemTray` in order to create a `TrayIcon` object.

### 2.9.3 Popup Menu Issues

In the `JPopupMenu.setInvoker()` method, the invoker is the component in which the popup menu is to be displayed. If this property is set to null, the popup menu does not function correctly.

The solution is to set the popup's invoker to itself.

### 2.9.4 Background/Foreground Color Inheritance

Many AWT components use their own defaults for background and foreground colors instead of using the colors of their parents.

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.

To ensure the consistency of your application on every platform, use explicit color assignment (both foreground and background) for every component or container.

### 2.9.5 AWT Panel Size Restriction

The AWT Container has a size limitation. On most platforms, this limit is 32767 pixels. This means that, for example, if the canvas objects are 25 pixels high, a Java AWT panel cannot display more than about 1400 objects.

Unfortunately there is no way to change this limit, either with Java code or with native code. The limit depends on what data type the operating system uses to store a widget size. For example, the Windows 2000/XP operating system and the Linux X operating system use `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.

Refer to the documentation for your platform for information.

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.

CHAPTER 3

# Troubleshooting 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, included in the following sections:

See Appendix A, "Java 2D Properties," for a summary of Java 2D properties.

See also the Java 2D FAQ (`http://java.sun.com/products/java-media/2D/reference/faqs/index.html`).

## 3.1 Changing Rendering Pipelines and Their Properties

Java 2D uses a set of pipelines, which can be roughly defined as different ways of rendering the primitives. These pipelines are as follows:

- X11 pipeline, which is the default for Solaris OS and Linux
- OpenGL pipeline, which is an alternative on Solaris OS and Linux, as well as Windows
- DirectDraw/GDI pipeline, which is the default on Windows
- Direct3D pipeline, which is an alternative on Windows

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.

The following procedure can help you troubleshoot Java 2D issues:

1. Determine the default pipeline used in your configuration.
2. Either change the pipeline to another one, or modify the properties of the default pipeline.
3. If the problem disappears, you have found a workaround.

4. If the problem persists, try changing another property or pipeline.

This section describes how to change the Java 2D pipelines and their properties for each operating system: UNIX (Solaris OS and Linux), and Windows.

For the default X11 pipeline on Solaris OS and Linux, the following information is provided:

- Properties related to the use of X11 pixmaps
  - Disabling the use of pixmaps by the X11 pipeline
  - Disabling/forcing the use of shared memory pixmaps by the X11 pipeline
- Use of Shared Memory Extension by the X11 pipeline
  - Increasing the amount of shared memory available to X server and Java 2D
  - Disabling the use of Shared Memory Extension by the X11 pipeline
- Solaris OS on SPARC: use of DGA in certain configurations
  - Detecting if the DGA extension is used for rendering to the screen
  - Typical issues caused by the use of DGA
  - Controlling the use of DGA by Java 2D
- Solaris OS on SPARC: changing the default visual used by Java 2D

For the default DirectDraw/GDI pipeline on Windows, the following information is provided:

- Disabling the use of DirectDraw
- Forcing the use of DirectDraw pipeline
- Disabling the built-in punting mechanism
- Disabling the use of DirectDraw blit operations

For the alternative Direct3D pipeline on Windows, the following information is provided:

- Disabling the Direct3D pipeline
- Forcing the use of the Direct3D pipeline
- Diagnosing rendering problems with the Direct3D pipeline

This section also provides information on using the OpenGL pipeline as an alternative pipeline on Solaris OS, Linux, and Windows.

### 3.1.1 Solaris OS and Linux: Using Default X11 Pipeline

On Unix platforms the default pipeline is the X11 pipeline. This pipeline uses the X protocol for rendering to the screen or to certain types of offscreen images, such as `VolatileImages`, or "compatible" images (images that are created with the `GraphicsConfiguration.createCompatibleImage()` method). Such 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.

#### 3.1.1.1 Properties Related to the Use of X11 Pixmaps

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. Such cases typically involve the use of operations which 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 a system memory (over the network in the case of remote X server), performing the rendering, and then sending the pixels back to the pixmap. Such operations could result in extremely poor performance, especially if the X server is remote.

##### Disabling the Use of Pixmaps by the X11 Pipeline

To disable the use of pixmaps by Java2D, pass the following property to the Java VM: `-Dsun.java2d.pmoffscreen=false`.

##### Disabling/Forcing the Use of Shared Memory Pixmaps by the X11 Pipeline

To minimize the effect of such operations requiring reading of pixels from a pixmap on overall performance, the X11 pipeline uses shared memory pixmaps for storing images which are often read from. Note that 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 disablng them, or forcing all images to be always stored in shared memory pixmaps.

- 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`.

First try forcing the shared memory pixmaps, as it is 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.

#### 3.1.1.2 Use of MIT Shared Memory Extension by the X11 Pipeline

The Java 2D X11 pipeline uses by default the MIT Shared Memory Extension (MIT SHM). This extension allows a faster exchange of data between the client (the Java application) and the X server, and this can significantly improve the performance of Java applications.

##### Increasing Shared Memory Available to X Server and Java 2D

On Solaris OS releases 8 and earlier it was sometimes necessary to increase the amount of shared memory available to the system (and to X server in particular) as the default was too low, resulting in poor rendering performance. Increasing the amount of shared memory and shared memory segments can result in better performance.

To change the default settings on Solaris OS, edit the `/etc/system` file, changing the `shmsys:shminfo_*` settings, as in the following example. Note that this is not needed on Solaris 9 OS and newer.

```
set shmsys:shminfo_shmmax=10000000
set shmsys:shminfo_shmni=200
set shmsys:shminfo_shminfo=150
```

On Linux this setting can be configured by editing the `/proc/sys/kernel/shm*` files.

##### Disabling Use of Shared Memory Extension by the X11 Pipeline

In case of problems (such as crashes, or rendering artifacts) with older X servers and 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`.

#### 3.1.1.3 Solaris OS on SPARC: Use of DGA in Certain Configurations

On SPARC hardware, if the framebuffer supports Sun's DGA (Direct Graphics Access) X server extension, and Java 2D has a corresponding module for accessing the framebuffer, DGA will be used for rendering to the screen.

All offscreen images will reside in Java heap memory, and Java 2D's software-only rendering pipeline is used for rendering to them. This is different from a typical UNIX configuration, where X11 pixmaps are used for offscreen images.

##### Detecting Use of DGA Extension for Rendering to Screen

To detect if the DGA extension is used for rendering ot the screen, run any Java application which does some rendering or displays a GUI, and check if a `/tmp/wg*` file was created when the application started. Exit the application and verify that the file has been deleted. If this is the case, then on this system Java 2D is using DGA.

##### Typical Issues Caused by Use of DGA

Since DGA allows direct access to the framebuffer's video memory, the typical problems include corruption outside of window bounds, complete system, and X server lock-ups.

##### Controlling Use of DGA by Java 2D

If it is determined that DGA is being used, the first thing to try is to disable it. This can be done by setting the `NO_J2D_DGA` environment variable to `true`. This forces the default UNIX path to use only X11 for rendering to the screen, and pixmaps for accelerating offscreen images.

In some cases it could be beneficial to enable the use of pixmaps, while also using DGA for rendering to the screen. To force the use of pixmaps for accelerating offscreen images, set the following property when starting the application: `-Dsun.java2d.pmoffscreen=true`.

## 3.1.2 Solaris OS on SPARC: Changing Default Visual Used by Java 2D

On certain video boards on the SPARC platform, more than one visual can be available from the X server. By default Java 2D tries to select the best visual, where "best" is typically a higher-bit depth visual. For example, on some Solaris OS releases the default X11 visual is 8-bit PseudoColor, although 24-bit visual is also available. In such cases Java 2D will select a 24-bit TrueColor visual as the default for Java windows.

While it is possible to create a Java top-level window with a `GraphicsConfiguration` object corresponding to a different visual, in some cases it is necessary to make Java use a different default visual instead. This can be done by setting the `FORCEDEFVIS` environment variable. It can be set to `true` to force the use of the default X server visual (even if it is not the best one), or it can be set to a hexadecimal number corresponding to the visual ID as reported by tools like `xdpyinfo`.

To determine your X server default visual, execute the `xdpyinfo` command and look at the `default visual id` field.

## 3.1.3 Windows OS: Using Default 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.

Starting with the Java SE 6 release, when the application enters full-screen mode, the new Direct3D pipeline can be used, if the drivers satisfy the requirements. 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.

### 3.1.3.1 Disabling the Use of DirectDraw

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 to with the default software pipeline. All onscreen rendering, as well as copies of offscreen images to the screen will be performed using GDI.

### 3.1.3.2 Forcing the Use of DirectDraw Pipeline

In case the pipeline was disabled by default for some reason, 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.

### 3.1.3.3 Disabling the Built-in Punting Mechanism

In general, the DirectDraw pipeline attempts to place the offscreen surfaces in the framebuffer's video memory, which provides fast copies from these surfaces to the screen or other accelerated surfaces, as well as hardware accelerated rendering of certain graphics operations.

However, if the pipeline cannot perform an operation using the DirectDraw API (operations using, for example, alpha compositing, or transforms, or antialiasing), the rendering is performed using the software pipeline. In some cases this means that the pixels of the destination surface, which resides in VRAM, need to be read into system memory, which is a very expensive operation.

To limit the impact of unaccelerated rendering to VRAM-based surfaces, there exists a punting mechanism, which moves the surface which 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 the video memory.

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 that this can result in performance degradation, as the software loops may be reading pixels from VRAM on each operation. In this case you may consider disabling the DirectDraw pipeline (see above).

#### 3.1.3.4 Disabling the DirectDraw Blit Operations

In a blit operation (Bit Block Transfer), two bitmap patterns are combined. This operation basically 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 that this 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.

### 3.1.4 Windows OS: Using Direct3D Pipeline (Full-Screen Mode)

Starting with the Java SE 6 release, the Direct3D pipeline uses the Direct3D API for rendering. This pipeline is enabled in full-screen mode by default, if the drivers support the required features and the level of rendering quality.

With both the Java SE 5 and 6 releases, it is possible to enable the Direct3D pipeline or to force its use, as described in the subsections below.

Consider enabling the Direct3D pipeline for your application if it makes heavy use of 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.

#### 3.1.4.1 Disabling 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 such cases, with both the Java SE 5 and 6 releases, pass the parameter `-Dsun.java2d.d3d=false` to the Java VM, or set the `J2D_D3D` environment variable to `false`.

#### 3.1.4.2 Forcing the Use of the Direct3D Pipeline

With both the Java SE 5 and 6 releases, 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 that the pipeline is enabled only if the drivers support minimum required features.

#### 3.1.4.3 Diagnosing Rendering Problems with Direct3D Pipeline

With the Java SE 6 release, 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`.

Refer to 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 very likely to be a video driver bug.

The `rgb` rasterizer is available only if the Direct3D SDK is installed. This SDK can be obtained from Microsoft Game Technologies Center (`http://msdn.microsoft.com/directx/`).

For performance or quality problems with text rendering with the Direct3D pipeline, you can force the use of 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`.

### 3.1.5 Using OpenGL Pipeline (SolarisOS, Linux, and Windows)

The OpenGL pipeline was first made available in the J2SE 5.0 release on Solaris OS, Linux, and Windows. This alternate pipeline uses the hardware-accelerated, cross-platform OpenGL API when rendering to `VolatileImages`, to backbuffers created with `BufferStrategy` API, and to the screen.

This pipeline can offer great performance advantages over the default (X11 or GDI/DirectDraw) pipelines for certain applications. Consider enabling the pipeline for your application if it makes heavy use of rendering operations like alpha compositing, antialiasing, and transforms.

For a complete list of Java 2D operations that are accelerated by the OpenGL pipeline, refer to the article Behind the Graphics2D: The OpenGL-based Pipeline (`http://today.java.net/cs/user/print/a/147`).

### 3.1.5.1 Enabling the OpenGL Pipeline

The OpenGL pipeline is currently 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"):

```
-Dsun.java2d.opengl=True
```

### 3.1.5.2 Minimum Requirements

The OpenGL pipeline will not be enabled if the hardware or drivers do not meet the minimum requirements. If for some reason one of the following requirements is not met, Java 2D will fall back and use the default pipeline (X11 on Solaris/Linux, GDI/DirectDraw on Windows), which means your application will continue to work correctly, but without the OpenGL acceleration.

The minimum requirements for Solaris OS and Linux 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

### 3.1.5.3 Latest OpenGL Drivers

Since 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 | Web Site | Platforms | Cards Known to Work |
|---|---|---|---|
| ATI | http://ati.com | Linux, Windows | Radeon 8500 and above, FireGL series |
| Nvidia | http://nvidia.com | Solaris OS on x64, Linux, Windows | GeForce 2 series and above, Quadro FX series and above |
| Sun | http://sun.com | Solaris OS on SPARC | Expert3D series, XVR-500, XVR-600, XVR-1200, XVR-2500 |
| Xi Graphics | http://xig.com | Solaris OS on x86, Linux | Various (check with Xi Graphics) |

#### 3.1.5.4 Diagnosing Startup Issues

As mentioned above, the OpenGL pipeline might not be enabled on certain machines for various reasons. For example, the drivers might not be properly installed and might report an insufficient version number. Alternatively, your machine might have an older graphics card that does not support the appropriate OpenGL version or extensions.

In the Java SE 6 release, 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 follows.

On Windows:

```
# set J2D_TRACE_LEVEL=4
# java -Dsun.java2d.opengl=True YourApp
```

On Solaris OS and Linux:

```
# export J2D_TRACE_LEVEL=4
# java -Dsun.java2d.opengl=True YourApp
```

The output will be different depending on your platform and the installed graphics hardware, but it can give you some insight into the reasons why the OpenGL pipeline is not being successfully enabled for your configuration. Note that this output is especially useful when filing bug reports intended for the Java 2D team at Sun, as discussed below.

#### 3.1.5.5 Diagnosing Rendering and Performance Issues

Since the OpenGL pipeline relies so heavily on the underlying graphics hardware and drivers, it might sometimes be difficult to determine whether rendering or performance issues are being caused by Java 2D or by the OpenGL drivers.

One feature new to the OpenGL pipeline in the Java SE 6 release is the use of the `GL_EXT_framebuffer_object` extension, which provides better performance for rendering and reduced VRAM consumption when using `VolatileImages`. This “FBO“ codepath is enabled by default when the OpenGL pipeline is enabled, but only if your graphics hardware and driver support this OpenGL extension. This extension is generally available on Nvidia GeForce/Quadro FX series and newer, and on ATI Radeon 9500 and newer. If you suspect that the "FBO" codepath is causing problems in your application, you can disable it by setting the following system property:

```
-Dsun.java2d.opengl.fbobject=false
```

Setting this property will cause Java 2D to fall back on the older "pbuffer-based" codepath.

If you find that a certain Java 2D operation causes different visual results with the OpenGL pipeline enabled than without, 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, 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 Chapter 8, "Submitting Bug Reports"). When filing bug reports, be as detailed as possible, and always 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 previous section)
- If on Solaris OS or Linux, the output of the `glxinfo` command

## 3.2 Generic Performance Issues

This section contains the following subsections:

### 3.2.1 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 which can be hardware-accelerated are `VolatileImage`, `screen`, and `BufferStrategy`. If a destination is accelerated, rendering which goes to such 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. Such images are known as "managed images."

Ideally, all operations performed to an accelerated surface are hardware-accelerated. In this case the application takes the full advantage that 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.

### 3.2.2 Using Java 2D 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. See the description of the `trace` property at System Properties for Java 2D Technology (`http://java.sun.com/javase/6/docs/technotes/guides/2d/flags.html`).

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 that 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()` returns `Translucency.TRANSLUCENT`), or 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 alpha value which 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 the tracing and make sure you do not use unaccelerated primitives unless they are needed.

## 3.2.3 Causes of Poor Rendering Performance

Some of the possible causes of poor rendering performance are described in the following subsections:

- "3.2.3.1 Mixing Accelerated and Non-accelerated Rendering" on page 54
- "3.2.3.2 Using Non-optimal Rendering Primitives" on page 54
- "3.2.3.3 Using Heap-based Destination Surface (a `BufferedImage`)" on page 54
- "3.2.3.4 Defeating Built-in Acceleration Mechanisms" on page 55

#### 3.2.3.1 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, 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 above.

Note, however, that 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, 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 that this property must be set prior to any GUI-related operations because it is read only once.

#### 3.2.3.2 Using 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.

#### 3.2.3.3 Using Heap-based Destination Surface (a `BufferedImage`)

Rendering to a `BufferedImage` almost always uses software loops.

An exception is that on some SPARC systems the VIS instruction set may be used for accelerating certain imaging operations. Refer to web site for the VIS Instruction Set (`http://www.sun.com/processors/vis/`).

To ensure that the rendering has the opportunity of being hardware accelerated, choose a `BufferStrategy` or a `VolatileImage` object as the rendering destination.

### 3.2.3.4 Defeating Built-in Acceleration Mechanisms

Java 2D attempts to accelerate certain types of images. Contents of images may be cached in video memory for faster copying to accelerated destinations such as `VolatileImages`. These mechanisms can be unknowingly defeated by the application, for example, in the following cases:

- "Getting Direct Access to the Pixels with `getDataBuffer()`" on page 55
- "Rendering to a Sprite Before Every Copy" on page 55
- "Exhausting Accelerated Memory Resources" on page 55

#### Getting Direct Access to the Pixels with `getDataBuffer()`

If an application gets access to `BufferedImage` pixels by using the `getRaster().getDataBuffer()` API, 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 such image.

There are two ways to avoid this problem:

- If possible, do not call `getDataBuffer()`. Instead, work with `WriteableRaster`, which can be obtained with the `BufferedImage.getRaster()` method.
- If you do need to modify the pixels directly, 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.

#### Rendering to a Sprite Before Every Copy

If an application renders to an image every time before copying it to an accelerated surface (`VolatileImage`, `BufferStrategy`), the image cannot take advantage of being cached in accelerated memory. This is because the cached copy has to 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.

#### Exhausting Accelerated Memory Resources

If the application uses many images, it may exhaust the available accelerated memory. If this is indeed the cause of performance issues for your application, 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 in the following ways:

- Do not hold 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.
- 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.

## 3.2.4 Improving 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, there are certain approaches to improving performance:

- "3.2.4.1 Use of Image Types or Operations With Optimized Support" on page 56
- "3.2.4.2 Transparency vs Translucency" on page 56

### 3.2.4.1 Use of 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.

### 3.2.4.2 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).`

## 3.3 Text-Related Issues

This section describes some issues that can be related to text rendering, included in the following subsections:

### 3.3.1 Application Crash During Text Rendering

If an application crashes during text rendering, first check the fatal error log file. See Appendix B, "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. Here is an example of 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. Here is an example of typical output.

```
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 be in fact innocent. 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 above output.

Another verification approach is to use the Font2DTest tool (`demo/jfc/Font2DTest`) to test fonts that you suspect. You can specify a particular font size, style, and rasterization mode. If the process of viewing a particular font with Font2DTest causes the JDK to crash, then it is very likely that it is the font that is causing the problems.

If you found a font causing the JDK to crash, it is very important to report this problem, including the particular font and the operating system, at `bugs.sun.com`. See Chapter 8, "Submitting Bug Reports," for more information about reporting bugs.

## 3.3.2 Differences in Text Appearance

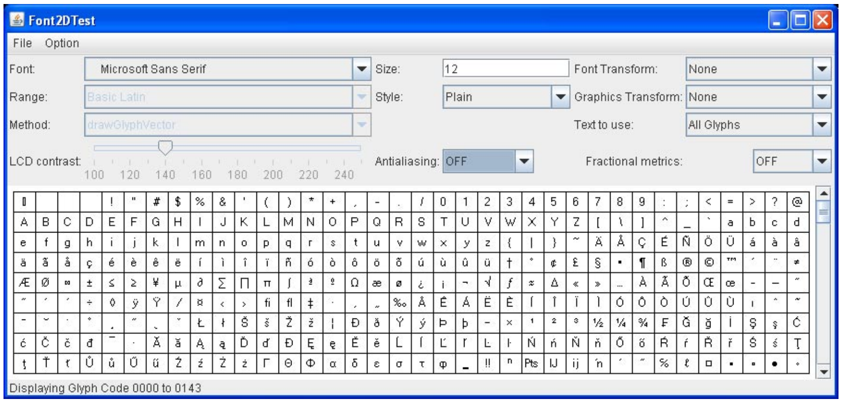
Java has its own font rasterizer, and you can expect some small differences between the appearance of text in a Java application and in a native application.

One of the most 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 remote X11 antialiasing is not enabled by default for performance reasons. For information about how to force antialiasing, see the Font and Test questions in the Java 2D FAQ (`http://java.sun.com/products/java-media/2D/reference/faqs/index.html#Font_and_Text_questions`).
- CJK fonts that use embedded bitmaps may render using the bitmaps instead of subpixel text.
- Some variants of unsupported desktops do not report their font smoothing settings properly. For example, KDE is unsupported but should generally work; however, some problem seems to prevent JDK from picking up the setting.

The best way to ensure that the configuration is what you expect is to run Font2DTest, explicitly select the font used by the native application, and set other parameters as appropriate. Here is a sample screen from the Font2DTest tool.



Font: Microsoft Sans Serif
Size: 12
Font Transform: None
Range: Basic Latin
Style: Plain
Graphics Transform: None
Method: drawGlyphVector
Text to use: All Glyphs
LCD contrast:
Antialiasing: OFF
Fractional metrics: OFF

Displaying Glyph Code 0000 to 0143

FIGURE 3–1 Sample Screen from Font2DTest Tool

Hint: you can input your own string by choosing "user text" in the drop-down menu labelled "Text to use."

The size of the font in the Java language is always expressed with 72 dpi. A native OS can use a different screen dpi, and therefore an adjustment needs to 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 feels, such as the Windows look and feel or the GTK look and feel (for Solaris OS and Linux), Swing components perform this adjustment automatically, but if you are running Font2DTest, the text display area will always use 72 dpi.

On operating systems other than Windows, the general recommendation is to use TrueType fonts instead of Type1 fonts. The easiest way to discover 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.

## 3.3.3 Metrics

If you find that text bounds are different from what you expect, 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 "width." For more details, refer to the Font and Text questions in the Java 2D FAQ (`http://java.sun.com/products/java-media/2D/reference/faqs/index.html#Font_and_Text_questions`).

## 3.4 Java 2D Printing

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

See also the Printing questions in the Java 2D FAQ (`http://java.sun.com/products/java-media/2D/reference/faqs/index.html#Printing_questions`).

**Issue: On Windows, JRE crashes during printing.**
*Cause:* The JRE uses Windows printer drivers and they might have problems.

*Solution:* Upgrade the Windows printer driver for the printer that is being used.

**Issue: On Windows, the printing seems to be successful, but the job does not print.**
*Cause:* Some jobs fail to properly spool to the printer.

*Solution:* In the printer driver properties, disable Advanced Printing Options.

**Issue: On Windows, the print dialog takes a long time to appear.**
*Cause:* Applications might cause the JRE to probe all printers, including those that are disconnected.

*Solution:* Look for disconnected or unreachable network printers and remove them from the list of printers.

**Issue: On Solaris OS and Linux, PrinterJob.printDialog() shows the error "No services found".**
*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 mentioned above.

◆ ◆ ◆ CHAPTER 4

# Troubleshooting 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.

## 4.1 General Debugging Tips for Swing

Swing's painting infrastructure changed quite extensively in Java SE 6. If you notice painting artifacts specific to Java SE 6 you can try turning off the new functionality. This can be done with the property `swing.bufferPerWindow`.

When you are debugging the Swing code which is executed while any menu is popped up, it is recommended to use the debugger remotely. Otherwise, the debugging process and the application execution block each other, and this prevents further work with the system. If that happens, the only action that can be taken is to kill the X server for Linux and Solaris. For more information, see the following bug in the Bug Database.

`http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6517045`

Some common Swing problems:

- Painting problems.
- Renderers.
- Updating models from wrong thread.
- Hangs.
- Responsiveness.
- Repainting issues.
- `isOpaque` usage.
- Startup: could be caused by small heap, loading unnecessary classes.

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.

Identifying bad renderers:

- Sluggish application, especially when scrolling.
- Use an optimizer to watch painting calls, look for calls to `getTableCellTRendererComponent`.

## 4.2 Specific Debugging Tips for Swing

The following subsections present some tips for troubleshooting Swing problems.

### 4.2.1 Incorrect Threading

Random exceptions and painting problems are usually the result of incorrect threading usage of Swing. *All* access to Swing components, unless specifically noted in the javadoc, must be done on the event dispatch thread. This includes any models (`TableModel`, `ListModel`, and others) that are attached to Swing components.

The best way to check for bad usage of Swing is by way of an instrumented `RepaintManager`, as illustrated by the following code:

```
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();
         }
     }
}
```

### 4.2.2 Overlapping Children of a `JComponent`

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`, components can randomly appear on top of others, depending upon which one repaint was invoked on.

### 4.2.3 Updating the Display

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, you must invoke repaint and possibly revalidate whenever the display or sizing information has been 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, it implies that the component did not invoke repaint or revalidate correctly.

### 4.2.4 Changing the Model

Just as you do not need to invoke repaint when you change a visible property of a Swing component, you also 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, a repaint event may not even work. In particular this will not work with `JTree`. The correct thing to do is to send out the appropriate model notification. This can usually be diagnosed by again resizing the window and noticing that the display has not updated correctly.

### 4.2.5 Adding or Removing Components

When you add or remove components, you need to invoke repaint or revalidate. Swing and AWT will not invoke repaint or revalidate in these situations, and therefore you must invoke them yourself.

### 4.2.6 Overriding Opaque

Another possible area of painting problems is if a component does not override opaque. Here is the documentation warning on this:

> Further, if you do not invoker super's implementation you must honor the opaque property, that is if this component is opaque, you must completely fill in the background in a non-opaque color. If you do not honor the opaque property 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.

### 4.2.7 Permanent Changes to a Graphics

Do not make any permanent changes to a Graphics passed to `paint`, `paintComponent`, or `paintChildren`. Here is the documentation warning on this:

> If you override this in a subclass you should not make permanent changes to the passed in Graphics. 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 from the passed in Graphics and manipulate it.

Not honoring this restriction will result in clipping or other weird visual artifacts.

### 4.2.8 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, you may lose double buffering.

### 4.2.9 Opaque Content Pane

Swing's painting architecture requires an opaque content pane. Here is the documentation:

> The painting architecture of Swing requires an opaque `JComponent` to exist in the containment hierarchy above all other components. This is typically provided by way of the content pane. If you replace the content pane, it is recommended that you make the content pane opaque by way of `setOpaque(true)`. Additionally, if the content pane overrides `paintComponent`, it will need to completely fill in the background in an opaque color in `paintComponent`.

### 4.2.10 Performance: Call to Renderer for Each Cell

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, there will be 1000 calls to the renderer.

### 4.2.11 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. For example, consider the following code:

```
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` installs on `myModel`. The solution is to invoke `table.setModel(new DefaultTableModel())`.

### 4.2.12 Mixing Heavyweight and Lightweight Components

Mixing heavyweight and lightweight components can work in certain scenarios, primarily 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 do use heavyweights, invoke the following methods:

- `JPopupMenu.setDefaultLightWeightPopupEnabled(false)`
- `ToolTipManager.sharedInstance().setLightWeightPopupEnabled(false)`

### 4.2.13 Tips for Using `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`.

### 4.2.14 Tracking Activity on Event Dispatch Thread

If a Swing application tries to do too much on the event dispatch thread, 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.

## 4.2.15 Differing Default Layout Managers

Problems can be caused by having 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`.

## 4.2.16 Listener Objects Dispatched to Deepest Component

`MouseListener` objects are dispatched to the deepest component that has `MouseListener` objects (or has enabled `MouseEvent` objects). A ramification of this is that 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, 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.

## 4.2.17 Adding a Component to Content Pane

Prior to J2SE 1.5 you could not add a component to a `JFrame`, `JWindow`, `JDialog` or `JApplet`. Instead, you needed to add the component to the content pane. As of J2SE 1.5 it is still the case that 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 these classes redirect to the content pane. In other words, doing `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 impacts `LayoutManagers` that only work with one component, such as `GroupLayout` and `BoxLayout`. For example, new `GroupLayout(frame)` will not work; instead, you need to do new `GroupLayout(frame.getContentPane()).`

### 4.2.18 Drag and Drop Support in Swing

When using Swing you should use Swing's drag–and–drop support as provided by `TransferHandler`. Otherwise, you will have to manage the selection and various other issues.

### 4.2.19 One Parent at a Time for a Component

Remember that a component can only exist in one parent at a time. Problems occur when you attempt to share menu items between menus. For example, `JMenuItem` is a component, and therefore can exist in only one menu at a time.

### 4.2.20 Problem With `JFileChooser` and Shortcuts on Windows

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, perform the following procedure:

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 press 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 the first step.

◆ ◆ ◆ CHAPTER 5

# Internationalization

This chapter provides information and guidance on troubleshooting issues that might be found in the area of internationalization support. For detailed information, visit the Java Internationalization site (`http://java.sun.com/javase/technologies/core/basic/intl/`).

## 5.1 Introduction

Before troubleshooting, make sure 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 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.

## 5.2 Troubleshooting

The Java Internationalization FAQ document (`http://java.sun.com/javase/technologies/core/basic/intl/faq.jsp`) answers general questions, as well as specific questions such as the following:

- Why doesn't my application display any Chinese, Japanese, or Korean characters even though I have fonts for these languages installed?
- Why can I see some characters in Swing components, but not in peered AWT components?
- Why can't my application display all Unicode characters even though I have a Unicode font installed?
- Why does the response character encoding in my JSP-based application change after I set it?

CHAPTER 6

# Troubleshooting Java Sound

This chapter describes some issues that can arise with the Java Sound technology and suggests causes and workarounds.

## 6.1 System Sound Configuration

Make sure that your audio system is correctly configured (sound card driver/DirectSound for Windows, ALSA or OSS for Linux, OSS or Audio Mixer for Solaris OS). 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 Solaris OS and Linux, 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.

## 6.2 Audio File Formats

Java Sound supports a set of audio file formats, for example `.au`, `.aif`, and `.wav`. Most of the file formats are only containers and can contain audio data in various compressed audio formats. Java Sound file readers support some formats (uncompressed PCM, a-law, mu-law), but do not support ADPCM, mp3, and others.

Java Sound also supports plug-ins for file readers and writers through the service provider interface (SPI). You can use Sun, third-party, or your own plug-ins to read various audio files. In any case you must handle the presence of the plug-in, for example, by distributing the required plug-ins with your application or by requiring plug-ins to be installed in the client Java environment.

## 6.3 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 if not all present sound cards. Most sound cards support only PCM formats, and even if the driver supports mu-law, it requires some modification to the software. If you need to play or record mu-law data, the preferred way is to convert it to PCM format through a format converter.

See the documentation for `AudioSystem.getAudioInputStream` for details about format conversion.

## 6.4 Overrun and Underrun Conditions

Recorded data is kept in a `DataLine` buffer. If you did not read from the line for a long time, an "overrun" condition will occur, and older data will be replaced with new. 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 has been written to the line, 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, a separate thread for writing data into the line is also the preferred method to use. If you use `Clip` for playing, the `Clip` implementation creates such a thread itself.

## 6.5 Default Soundbank Availability

Sometimes the `Synthesizer` class does not produce sound on some systems. By default JRE for Windows does not install a soundbank file, which is used by `Synthesizer` to produce sounds. You can download the soundbank file and install it, or reinstall JRE with the "additional media support" checkbox selected.

◆ ◆ ◆ CHAPTER 7

# Troubleshooting Java Plug-in

This chapter describes some issues that can arise with the Java Plug-in technology and suggests causes and workarounds.

## 7.1 Java Plug-in

The following issues might arise with Java Plug-in:

**Issue: Internet Explorer browser hangs when a frame containing a `while` loop in its constructor is about to be created.**

*Cause:* The code path for pumping messages to the browser is different for a modal and a non-modal dialog. In this particular case, the frame (though non-modal) behaves as a modal for some time after it enters into a `while` loop.

*Workaround:* Replace the frame with a modal dialog.

**Issue: Browser hangs in an applet.**

*Cause:* Determine the cause as follows:

- Add `-verbose` in the Java Control Panel.
- Restart the browser and run the applet again. When the applet is loaded, a dos box appears on the screen.
- Press Ctrl-\ or Ctrl-Break to get the stack trace when the applet gets hung.

*Workaround:* None

**Issue: Rendering artifacts/flickering issue while scrolling an applet in a browser (Internet Explorer).**

*Workaround:* Set the property `sun.awt.noerasebackground=true` in the Java control panel and restart the browser.

# CHAPTER 8

# Submitting Bug Reports

This chapter provides guidance on how to submit a bug report. It includes suggestions about what to try before submitting a report and what data to collect for the report.

## 8.1 Checking for Existing Fixes in Update Releases

The current platform is Java SE 6. Regularly scheduled updates to this 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 site (`http://java.sun.com/javase/downloads`).

The download site includes 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 include the list of fixes in previous update releases. If you encounter an issue, or suspect a bug, then, as an early step in the diagnosis, check the list of fixes that are available in the most recent update release.

Sometimes it is not obvious if an issue is a duplicate of a bug that is already fixed. Therefore, where possible, test with the latest update release to see if the problem persists.

## 8.2 Preparing to Submit a Bug Report

Before submitting a big report, consider the following recommendations:

- Collect as much relevant data as possible. For example, generate a thread-dump in the case of a deadlock, or locate the core file (where applicable) and `hs_err` file in the case of a crash. In all cases it is important to document the environment and the actions performed just before the problem is encountered.
- 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 when compared to test cases that consist of a large complex application.
- Search the bug database to see if this bug or a similar bug has been reported. If the bug has already been reported, the bug report might have further information, such as the following:
  - If the bug has already been fixed, the release in which it was fixed.
  - A workaround for the problem.
  - Comments in the evaluation that explain, in further detail, the circumstances that cause the bug to arise.

  The bug database is located at `http://bugs.sun.com/bugdatabase/index.jsp`.
- If you conclude that the bug has not already been reported, submit a new bug.

Before submitting a bug, verify that the environment where the problem arises is a supported configuration. See the Supported System Configurations site (`http://java.sun.com/javase/6/webnotes/install/system-configurations.html`).

In addition to the system configurations, check the list of supported locales. See the Supported Locales web page (`http://java.sun.com/javase/6/docs/technotes/guides/intl/locale.doc.html`).

In the case of the Solaris Operating System, check the recommended patch cluster for the operating system release to ensure that the recommended patches are installed. The recommended patch list can be obtained at thePatches & Updates from the Sun Update Connection (`http://sunsolve.sun.com/pub-cgi/show.pl?target=patches/JavaSE`)

## 8.3 Collecting Data for a Bug Report

In general it is recommended to collect as much relevant data as possible when you create a bug report or submit a support call. This section suggests the data to collect and, where applicable, it provides recommendations for the commands or general procedure for obtaining the data.

The following data can be collected prior to submitting a bug report:

- Hardware details
- Operating system details
- Java SE version information
- Command-line options
- Environment variables
- Fatal error log (in the case of a crash)
- Core or crash dump (in the case of a crash and possibly a hang)
- Detailed description of the problem, including test case (where possible)

- Logs or trace information (where applicable)
- Results from troubleshooting steps

The following sections present more detail for each type of data.

## 8.3.1 Hardware Details

Sometimes a bug arises or can be reproduced only on certain hardware configurations. If a fatal error occurs, the error log might contain the hardware details. If an error log is not available, 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.

## 8.3.2 Operating System

On the Solaris Operating System, the `showrev -a` command prints the operating system version and patch information.

On Linux, it is important to know which distribution and version is used. Sometimes the `/etc/*release` file indicates the release information, but as 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 older kernels and has “fixed stack” and “floating stack” variants. The Native POSIX Thread Library (`NTPL`) 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.

## 8.3.3 Java SE Version

The Java SE version string can be obtained using 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.

## 8.3.4 Command-Line Options

If the bug report does not include a fatal error log, 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 2GB 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.

## 8.3.5 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_HOME`
- `JRE_HOME`
- `JAVA_TOOL_OPTIONS`
- `_JAVA_OPTIONS`
- `CLASSPATH`
- `JAVA_COMPILER`
- `PATH`
- `USERNAME`

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

- On Solaris OS and Linux, collect the values of the following environment variables.
  - `LD_LIBRARY_PATH`
  - `LD_PRELOAD`
  - `SHELL`
  - `DISPLAY`
  - `HOSTTYPE`
  - `OSTYPE`
  - `ARCH`
  - `MACHTYPE`
- On the Linux operating system, collect the values of the following environment variables.
  - `LD_ASSUME_KERNEL`
  - `_JAVA_SR_SIGNUM`
- On the Windows operating system, collect the values of the following environment variables.

- `OS`
- `PROCESSOR_IDENTIFIER`
- `_ALT_JAVA_HOME_DIR`

## 8.3.6 Fatal Error Log

When a fatal error occurs, an error log is created. See Appendix B, "Fatal Error Log," for detailed information about this file.

The error log contains much information obtained at the time of the fatal error, such as version and environment information, details on the threads that provoked the crash, and so forth.

If the fatal error log is generated, be sure to include it in the bug report or support call.

## 8.3.7 Core or 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 "8.4 Collecting Core Dumps" on page 82.

## 8.3.8 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 most importantly the events leading up to the time when the problem was encountered.

- If the problem is reproducible, list the stepsthat are required to demonstrate the problem.
- If the problem can be demonstrated with a small test case, 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), provide details on where and how to obtain the library.

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 sole means 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 arises.

## 8.3.9 Logs and Traces

In some cases, 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 to 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 using Ctrl-\ on Solaris OS and Linux and Ctrl-Break on the Windows operating system.

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

### 8.3.10 Results from Troubleshooting Steps

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, 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, this gives 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.

## 8.4 Collecting Core Dumps

This section explains how to generate and collect core dumps (also known as crash 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 means of 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 as a minimum. Collecting non-truncated good core dump files is essential for postmortem analysis of the crash.

## 8.4.1 Collecting Core Dumps on Solaris OS

With the Solaris Operating System, unhandled signals such as a 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`. The user can configure the location and name of the core dump using the core file administration utility, `coreadm`. This procedure is fully described in the man page for the `coreadm` utility.

The `ulimit` utility is used to get or set the limitations on the system resources available to the current shell and its descendants. Use the `ulimit -c` command to check or set the core file size limit. Make sure that the limit is set to `unlimited`; otherwise the core file could be truncated. Note that `ulimit` is a Bash shell built-in command; on a C shell, use the `limit` command.

Ensure that any scripts that are used to launch the VM or your application do not disable core dump creation.

The `gcore` utility can be used to get a core image of running processes. This utility accepts a process id (pid) of the process for which you want to force 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` command. 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.

### 8.4.1.1 Using the `ShowMessageBoxOnError` Option on Solaris OS

A Java process can be started with the `-XX:+ShowMessageBoxOnError` command-line option. When a fatal error is encountered, the process prints a message to standard error and waits for a `yes` or `no` response from standard input. Below is an example of output when an unexpected signal occurs.

```
=======================================================================
Unexpected Error
-----------------------------------------------------------------------
SIGSEGV (0xb) at pc=0xfeba31ac, pid=8677, tid=2
Do you want to debug the problem?
To debug, run 'dbx - 8677'; then switch to thread 2
```

```
Enter 'yes' to launch dbx automatically (PATH must include dbx)
Otherwise, press RETURN to abort...
==============================================================================
```

Before answering `yes` or pressing RETURN, use the `gcore` utility to force a core dump. Then you can type `yes` to launch the `dbx` debugger.

#### 8.4.1.2 Suspending a Process using `truss`

In situations where it is not possible to specify the `-XX:+ShowMessageBoxOnError` option, you might be able to use the `truss` utility. This Solaris OS utility is used to trace system calls and signals. You can use this utility to suspend the process when it reaches a specific function or system call.

The following command shows how to use the `truss` utility to suspend a process when the exit system call is executed (in other words, the process is about to exit).

```
$ truss -t \!all -s \!all -T exit -p pid
```

When the process calls `exit`, it will be suspended. At this point, you can attach the debugger to the process or call `gcore` to force a core dump.

## 8.4.2 Collecting Core Dumps on Linux

On the Linux operating system, 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. Make sure that the limit is set to `unlimited`; otherwise the core file could be truncated. Note that `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` command. 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.

#### 8.4.2.1 Using the `ShowMessageBoxOnError` Option on Linux

A Java process can be started with the `-XX:+ShowMessageBoxOnError` command-line option. When a fatal error is encountered, the process prints a message to standard error and waits for a `yes` or `no` response from standard input. Below is an example of 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...
=======================================================================
```

Type `yes` to launch the `gdb` (GNU Debugger) interface, as suggested by the error report shown above. 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. Make sure that the `gdb gcore` command is supported in your versions of `gdb`. Look for `help gcore` in the `gdb` command prompt.

### 8.4.3 Reasons for Not Getting a Core File

The following list explains the major reasons that a core file might not be generated. This list pertains to both Solaris OS and Linux, unless specified otherwise.

- The current user does not have permission to write in the current working directory of the process.
- The current 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 NFS (Network File System), and NFS failed just at the time the core dump was about to be created.
- Solaris OS only: The `coreadm` tool has been used to configure the directory and name of the core file, but any of the above reasons apply for the configured directory or filename.

- The core file size limit is too low. Check your core file limit using the `ulimit -c` command (Bash shell) or the `limit -c` command (C shell). If the output from this command is not `unlimited`, the core dump file size might not be large enough. If this is the case, 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 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 Java VM generated code. These are the only cases in which HotSpot dumps 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 just 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.

### 8.4.4 Collecting Crash Dumps on Windows

On the Windows operating system there are three types of crash dumps.

- Dr. Watson logfile, which is a text error log file that includes faulting stack trace and a few other details.
- User minidump, which can be 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 it executes the command in the `Debugger` value. If the value for `Debugger` is a valid command, a message box is created with two buttons: OK and Cancel. If the user clicks OK, the program is terminated. If the user clicks Cancel, 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, the system automatically starts the debugger and does not generate a message box.

#### 8.4.4.1 Configuring Dr. Watson

The Dr. Watson debugger is used to create crash dump files. By default, the Dr. Watson debugger (`drwtsn32.exe`) is installed into the Windows system folder (`%SystemRoot%\System32`).

To install Dr. Watson as the postmortem debugger, run the following command.

```
drwtsn32 -i
```

To configure name and location of crash dump files, run `drwtsn32` without any options.

```
drwtsn32
```

In the Dr. Watson GUI window, make sure that the Create Crash Dump File checkbox is set and that the crash dump file path and log file path are configured in their respective text fields.

Dr. Watson may be configured to create a full dump using the registry. The registry key is as follows.

```
System Key: [HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\DrWatson]
Entry Name: CreateCrashDump
Value: (0 = disabled, 1 = enabled)
```

Note that 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.

#### 8.4.4.2 Forcing 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 but instead 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 process in a non-invasive manner (that is, read-only). Normally Windows terminates a process after a crash dump is obtained but with the non-invasive attach it is possible to obtain a crash dump and let the process continue. To attach the debugger non-invasively requires selecting the Attach to Process option and clicking the Noninvasive checkbox.

When the debugger is attached, a crash dump can be obtained using the following command.

```
.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 has been created correctly.

Both `userdump.exe` and `windbg` require the process id (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), it might be difficult to recognize the process. In that case you can use the`jps` command line utility as it lists the pids of the Java processes only.

As with Solaris OS and Linux, you can also use the `-XX:+ShowMessageBoxOnError` command-line option on Windows. When a fatal error is encountered, 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 for the case where the process appears to be hung.

◆ ◆ ◆ APPENDIX A

# Java 2D Properties

This appendix presents properties that can be useful in troubleshooting Java 2D.

## A.1 Properties on Solaris OS and Linux

The following table describes the default values of some useful properties on Solaris OS and Linux platforms.

**TABLE A–1** Java 2D Defaults on Solaris OS and Linux

| Setup | DGA | SHM | Pixmaps | OnScreen | OffScreen |
|---|---|---|---|---|---|
| Solaris SPARC with DGA support | On | On | Off | DGA/Software | Software |
| Solaris SPARC with no DGA, Solaris x86, Linux, SunRay, VNC | Off | On | On | X11/MITSHM | Shared/Server Pixmaps |
| J2SE 1.4 or greater: Remote X server, ssh | Off | Off | On | X11 | Server Pixmaps |
| J2SE 1.3.1 or less: Remote X server, ssh | Off | Off | Off | X11 | Software |

The following list explains how to change the defaults.

- The X11 pipeline is the default pipeline for Solaris OS and Linux. Change this default as follows:
  - `-Dsun.java2d.opengl=true` — Attempt to enable the OpenGL pipeline.
- The use of DGA is controlled as follows:
  - `NO_J2D_DGA` unset — Use DGA, if available.
  - `NO_J2D_DGA` set — Disable the use of DGA.

- MIT Shared Memory Extension (SHM) is controlled as follows:
  - To use Shared Memory Extension, if available, specify either one of the following properties:
    - `NO_J2D_MITSHM` unset
    - `J2D_USE_MITSHM=true`
  - To *not* use Shared Memory Extension, 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.

## A.2 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 DirectDraw pipeline. GDI will be used instead.
  - `-Dsun.java2d.noddraw=false` — Enable the use of DirectDraw pipeline.
  - `-Dsun.java2d.d3d=false` — Disable the use of Direct3D pipeline.
  - `J2D_D3D=false` — Disable the use of Direct3D pipeline.
  - `-Dsun.java2d.d3d=true` — Enable the use of Direct3D pipeline.
  - `J2D_D3D=true` — Enable the use of 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.

APPENDIX B

# Fatal Error Log

When a fatal error occurs, an error log is created with information and the state obtained at the time of the fatal error.

Note that the format of this file can change slightly in update releases.

This appendix contains the following sections.

## B.1 Location of Fatal Error Log

The product flag `-XX:ErrorFile=`*file* can be used to specify where the file will be created, where *file* represents the full path for the file location. The substring `%%` in the *file* variable is converted to `%`, and the substring `%p` is converted to the process ID 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, by default the file name is `hs_err_pid`*pid*`.log`, where *pid* is the process ID 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 Solaris OS and Linux 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, the value of the `TEMP` environment variable is used.

## B.2 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 on the thread that provoked the fatal error and thread's stack trace
- The list of running threads and their state
- Summary information about the heap
- The 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 details.

The error log is a text file consisting of the following sections:

- A header that provides a brief description of the crash. See "B.3 Header Format" on page 94.
- A section with thread information. See "B.4 Thread Section Format" on page 97.
- A section with process information. See "B.5 Process Section Format" on page 100.
- A section with system information. See "B.6 System Section Format" on page 105.

**Note –** Note that the format of the fatal error log described here is based on Java SE 6. The format might be different with other releases.

## B.3 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.

The following is a sample header from a crash.

```
#
# An unexpected error has been detected by Java Runtime Environment:
#
#  SIGSEGV (0xb) at pc=0x417789d7, pid=21139, tid=1024
#
# Java VM: Java HotSpot(TM) Client VM (1.6.0-rc-b63 mixed mode, sharing)
# Problematic frame:
# C  [libNativeSEGV.so+0x9d7]

#
# If you would like to submit a bug report, please visit:
#   http://java.sun.com/webapps/bugreport/crash.jsp
#
```

This example shows that the VM crashed on an unexpected signal. The next line describes the signal type, program counter (pc) that caused the signal, process ID and thread ID, as follows.

```
#  SIGSEGV (0xb) at pc=0x417789d7, pid=21139, tid=1024
      |      |           |             |         +--- thread id
      |      |           |             +------------- process id
      |      |           +--------------------------- program counter
      |      |                                        (instruction pointer)
      |      +--------------------------------------- signal number
      +---------------------------------------------- signal name
```

The next line contains the VM version (Client VM or Server VM), an indication whether the application was run in mixed or interpreted mode, and an indication whether class file sharing was enabled.

```
# Java VM: Java HotSpot(TM) Client VM (1.6.0-rc-b63 mixed mode, sharing)
```

The next information is the function frame that caused the crash, as follows.

```
# Problematic frame:
# C  [libNativeSEGV.so+0x9d7]
  |              +-- 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.
  +----------------- Frame type
```

In this example, the “C” frame type indicates a native C frame. The following table shows the possible frame types.

TABLE B–1 Thread 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. Here is an example of how the header looks 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 the text "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.

## B.4 Thread Section Format

This section contains information about the thread that just crashed. If multiple threads crash at the same time, only one thread is printed.

### B.4.1 Thread Information

The first part of the thread section shows the thread that provoked the fatal error, as follows.

```
Current thread (0x0805ac88):  JavaThread "main" [_thread_in_native, id=21139]
                    |             |         |            |          +-- ID
                    |             |         |            +------------- state
                    |             |         +-------------------------- name
                    |             +------------------------------------ type
                    +-------------------------------------------------- pointer
```

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`

The following table shows the important thread states.

**TABLE B–2** Thread States

| Thread State | Description |
|---|---|
| `_thread_uninitialized` | Thread is not created. This occurs only in the case of memory corruption. |
| `_thread_new` | Thread has been created but it has not yet started. |
| `_thread_in_native` | Thread is running native code. The error is probably a bug in 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. |

**TABLE B–2** Thread States *(Continued)*

| Thread State | Description |
|---|---|
| `..._trans` | If any of the above states is followed by the string `_trans`, 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.

## B.4.2 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 follows.

```
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 Solaris OS and Linux systems the signal number (`si_signo`) and signal code (`si_code`) are used to identify the exception, as follows.

```
siginfo:si_signo=11, si_errno=0, si_code=1, si_addr=0x00004321
```

## B.4.3 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 (IA32) processor.

```
Registers:
EAX=0x00004321, EBX=0x41779dc0, ECX=0x080b8d28, EDX=0x00000000
ESP=0xbfffc1e0, EBP=0xbfffc1f8, ESI=0x4a6b9278, EDI=0x0805ac88
EIP=0x417789d7, CR2=0x00004321, EFLAGS=0x00010216
```

The register values might be useful when combined with instructions, as described below.

## B.4.4 Machine Instructions

After the register values, the error log 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 that 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=0xbfffc1e0)
0xbfffc1e0:   00000000 00000000 0818d068 00000000
0xbfffc1f0:   00000044 4a6b9278 bfffd208 41778a10
0xbfffc200:   00004321 00000000 00000cd8 0818d328
0xbfffc210:   00000000 00000000 00000004 00000003
0xbfffc220:   00000000 4000c78c 00000004 00000000
0xbfffc230:   00000000 00000000 00180003 00000000
0xbfffc240:   42010322 417786ec 00000000 00000000
0xbfffc250:   4177864c 40045250 400131e8 00000000
Instructions: (pc=0x417789d7)
0x417789c7:   ec 14 e8 72 ff ff ff 81 c3 f2 13 00 00 8b 45 08
0x417789d7:   0f b6 00 88 45 fb 8d 83 6f ee ff ff 89 04 24 e8
```

## B.4.5 Thread Stack

Where possible, the next output in the error log is the thread stack. 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. It is important to note that in some fatal error conditions the stack may be corrupt, and in this case this detail may not be available.

```
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]
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]
C  [java.exe+0x14c0]
C  [java.exe+0x64cd]
C  [kernel32.dll+0x214c7]

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
```

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 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.

### B.4.6 Further Details

If the error occurred in the VM thread or in a compiler thread, then further details may be printed. 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 provoked the fatal error. The task is a compiler task and the HotSpot Client VM is 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.

## B.5 Process Section Format

The process section is printed after the thread section. It contains information about the whole process, including thread list and memory usage of the process.

### B.5.1 Thread List

The thread list includes the threads that the VM is aware of. 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. The output format follows.

```
=>0x0805ac88 JavaThread "main" [_thread_in_native, id=21139]
|      |         |        |             |             +----- ID
|      |         |        |             +------------------- state
|      |         |        |                                  (JavaThread only)
|      |         |        +--------------------------------- name
|      |         +------------------------------------------ type
|      +---------------------------------------------------- pointer
+------------------------------------------------------ "=>" current thread
```

An example of this output follows.

```
Java Threads: ( => current thread )
  0x080c8da0 JavaThread "Low Memory Detector" daemon [_thread_blocked, id=21147]
  0x080c7988 JavaThread "CompilerThread0" daemon [_thread_blocked, id=21146]
  0x080c6a48 JavaThread "Signal Dispatcher" daemon [_thread_blocked, id=21145]
  0x080bb5f8 JavaThread "Finalizer" daemon [_thread_blocked, id=21144]
  0x080ba940 JavaThread "Reference Handler" daemon [_thread_blocked, id=21143]
=>0x0805ac88 JavaThread "main" [_thread_in_native, id=21139]

Other Threads:
  0x080b6070 VMThread [id=21142]
  0x080ca088 WatcherThread [id=21148]
```

The thread type and thread state are described in "B.4 Thread Section Format" on page 97.

## B.5.2 VM State

The next information is the VM state, which indicates the overall state of the virtual machine. The following table describes the general states.

**TABLE B–3** 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)
```

## B.5.3 Mutexes and Monitors

The next information in the error log is a list of mutexes and monitors that are currently owned by a thread. These mutexes are VM internal locks rather than monitors associated with Java objects. Below 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
```

### B.5.4 Heap Summary

The next information is a summary of the heap. 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 start-up and a GC has not yet promoted any objects into the tenured generation. An example of this output follows.

```
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.
```

### B.5.5 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 in the case of large applications. The memory map can be very useful when debugging some crashes, as it can tell you what libraries are actually being used, their location in memory, as well as the location of heap, stack, and guard pages.

The format of the memory map is operating-system-specific. On the Solaris Operating System, the base address and library name are printed. On the Linux system the process memory map (`/proc/`*pid*`/maps`) is printed. On the Windows system, the base and end addresses of each library are printed. The following example output was generated on Linux/x86. Note that most of the lines have been omitted from the example for the sake of brevity.

```
Dynamic libraries:
08048000-08056000 r-xp 00000000 03:05 259171     /h/jdk6/bin/java
08056000-08058000 rw-p 0000d000 03:05 259171     /h/jdk6/bin/java
08058000-0818e000 rwxp 00000000 00:00 0
40000000-40013000 r-xp 00000000 03:0a 400046     /lib/ld-2.2.5.so
40013000-40014000 rw-p 00013000 03:0a 400046     /lib/ld-2.2.5.so
40014000-40015000 r--p 00000000 00:00 0
```
*Lines omitted.*

```
4123d000-4125a000 rwxp 00001000 00:00 0
4125a000-4125f000 rwxp 00000000 00:00 0
4125f000-4127b000 rwxp 00023000 00:00 0
4127b000-4127e000 ---p 00003000 00:00 0
4127e000-412fb000 rwxp 00006000 00:00 0
412fb000-412fe000 ---p 00083000 00:00 0
412fe000-4137b000 rwxp 00086000 00:00 0
```
*Lines omitted.*
```
44600000-46570000 rwxp 00090000 00:00 0
46570000-46610000 rwxp 00000000 00:00 0
46610000-46a50000 rwxp 020a0000 00:00 0
46a50000-46bb0000 rwxp 00000000 00:00 0
46bb0000-4a570000 rwxp 02640000 00:00 0
```
*Lines omitted.*

The format of a line in the above memory map is as follows.

```
40049000-4035c000 r-xp 00000000 03:05 824473  /jdk1.5/jre/lib/i386/client/libjvm.so
|<------------->|  ^      ^       ^     ^        |<----------------------------------->|
  Memory region    |      |       |     |                           |
                   |      |       |     |                           |
  Permission   --- +      |       |     |                           |
    r: read               |       |     |                           |
    w: write              |       |     |                           |
    x: execute            |       |     |                           |
    p: private            |       |     |                           |
    s: share              |       |     |                           |
                          |       |     |                           |
  File offset   ----------+       |     |                           |
                                  |     |                           |
  Major ID and minor ID of -------+     |                           |
  the device where the file             |                           |
  is located (i.e. /dev/hda5)           |                           |
                                        |                           |
  inode number  ------------------------+                           |
                                                                    |
  File name  -------------------------------------------------------+
```

In the memory map output, 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 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 in the example below.

```
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
```

## B.5.6 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. An example follows.

```
VM Arguments:
java_command: NativeSEGV 2

Environment Variables:
JAVA_HOME=/h/jdk
PATH=/h/jdk/bin:.:/h/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin:
     /usr/dist/local/exe:/usr/dist/exe:/bin:/usr/sbin:/usr/ccs/bin:
     /usr/ucb:/usr/bsd:/usr/etc:/etc:/usr/dt/bin:/usr/openwin/bin:
     /usr/sbin:/sbin:/h:/net/prt-web/prt/bin
USERNAME=user
LD_LIBRARY_PATH=/h/jdk6/jre/lib/i386/client:/h/jdk6/jre/lib/i386:
     /h/jdk6/jre/../lib/i386:/h/bugs/NativeSEGV
SHELL=/bin/tcsh
DISPLAY=:0.0
HOSTTYPE=i386-linux
```

```
OSTYPE=linux
ARCH=Linux
MACHTYPE=i386
```

Note that 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.

### B.5.7 Signal Handlers

On Solaris OS and Linux, the next information in the error log is the list of signal handlers.

```
Signal Handlers:
SIGSEGV: [libjvm.so+0x3aea90], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGBUS: [libjvm.so+0x3aea90], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGFPE: [libjvm.so+0x304e70], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGPIPE: [libjvm.so+0x304e70], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGILL: [libjvm.so+0x304e70], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGUSR1: SIG_DFL, sa_mask[0]=0x00000000, sa_flags=0x00000000
SIGUSR2: [libjvm.so+0x306e80], sa_mask[0]=0x80000000, sa_flags=0x10000004
SIGHUP: [libjvm.so+0x3068a0], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGINT: [libjvm.so+0x3068a0], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGQUIT: [libjvm.so+0x3068a0], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGTERM: [libjvm.so+0x3068a0], sa_mask[0]=0xfffbfeff, sa_flags=0x10000004
SIGUSR2: [libjvm.so+0x306e80], sa_mask[0]=0x80000000, sa_flags=0x10000004
```

## B.6 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 Solaris 9 OS system.

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

OS:                       Solaris 9 12/05 s9s_u5wos_08b SPARC
           Copyright 2005 Sun Microsystems, Inc.  All Rights Reserved.
                        Use is subject to license terms.
                           Assembled 21 November 2005

uname:SunOS 5.9 Generic_112233-10 sun4u  (T2 libthread)
rlimit: STACK 8192k, CORE infinity, NOFILE 65536, AS infinity
load average:0.41 0.14 0.09

CPU:total 2 has_v8, has_v9, has_vis1, has_vis2, is_ultra3
```

```
Memory: 8k page, physical 2097152k(1394472k free)

vm_info: Java HotSpot(TM) Client VM (1.5-internal) for solaris-sparc,
built on Aug 12 2005 10:22:32 by unknown with unknown Workshop:0x550
```

On Solaris OS and Linux, the operating system information is contained 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 Solaris OS the `uname` system call is used to get the name for the kernel. The thread library (T1 or T2) is also printed.

On the Linux system the `uname` system call is also used to get the kernel name. The `libc` version and the thread library type are also printed. An example follows.

```
uname:Linux 2.4.18-3smp #1 SMP Thu Apr 18 07:27:31 EDT 2002 i686
libc:glibc 2.2.5 stable linuxthreads (floating stack)
    |<- glibc version ->|<--  pthread type       -->|
```

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 an 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 Solaris OS and Linux, the next information is the `rlimit` information. Note that the default stack size of the VM is usually smaller than the system limit. An example follows.

```
rlimit: STACK 8192k, CORE 0k, NPROC 4092, NOFILE 1024, AS infinity
             |          |         |           |           virtual memory (-v)
             |          |         |           +--- max open files (ulimit -n)
             |          |         +----------- max user processes (ulimit -u)
             |          +------------------------ core dump size (ulimit -c)
             +---------------------------------------- stack size (ulimit -s)
load average:0.04 0.05 0.02
```

The next information specifies the CPU architecture and capabilities identified by the VM at start-up, as in the following example.

```
CPU:total 2 family 6, cmov, cx8, fxsr, mmx, sse
         |        | |<----- CPU features ---->|
         |        |
         |        +--- processor family (IA32 only):
         |              3 - i386
         |              4 - i486
         |              5 - Pentium
         |              6 - PentiumPro, PII, PIII
         |             15 - Pentium 4
         +------------ Total number of CPUs
```

The following table shows the possible CPU features on a SPARC system.

**TABLE B–4** 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. |

The following table shows the possible CPU features on an Intel/IA32 system.

**TABLE B–5** 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. |

The following table shows the possible CPU features on an AMD64/EM64T system.

TABLE B–6 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 follows.

```
                                                        unused swap space
                                total amount of swap space      |
                      unused physical memory         |          |
  total amount of physical memory     |              |          |
       page size            |         |              |          |
           v                v         v              v          v
Memory: 4k page, physical 513604k(11228k free), swap 530104k(497504k free)
```

Some systems require swap space to be at lease twice the size of real physical memory, whereas other systems do not have any such requirements. As a general rule, if both physical memory and swap space are almost full, there is good reason to suspect that the crash was due to insufficient memory.

On Linux systems 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 does mean the amount of unused physical memory reported by 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.