Troubleshooting Guide for Java SE 7 Desktop Technologies

ORACLE

Chapter 2

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 the `sun.awt.noisyerrorhandler` system property to `true`. In Java SE 6 and before, the `NOISY_AWT` environment variable was used for this purpose.
- 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.

Prior to Java SE 7, exceptions thrown in the AWT Event Dispatch Thread (EDT) could be caught by setting the system property `sun.awt.exception.handler` to the name of a class that implements a `public void handle(Throwable)` method. This mechanism has been updated in Java SE 7 to use the standard `Thread.UncaughtExceptionHandler` interface.

Loggers can produce helpful output when debugging AWT problems. For information on using loggers, consult the Java Logging Overview and the `java.util.logging` package description.

The following loggers are available:

- java.awt
- java.awt.focus
- java.awt.event
- java.awt.mixing
- sun.awt
- sun.awt.windows
- sun.awt.X11

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 latest Java SE 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 or later, 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 7 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 7 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.

2.7.2 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, a concept of "synthetic focus" has been implemented. It means that a focus owner component only emulates its focusable state whereas real native focus is set to a "focus proxy" component. This component receives key and input method native messages and dispatches them to a focus owner. Prior to JDK7 a focus proxy component was a dedicated hidden child component inside a frame/dialog. In JDK7 a frame/dialog itself serves as a focus proxy. Now it proxies focus not only for components in an owned window but for all child components as well. A simple window never receives native focus and relies on focus proxy of its owner. This mechanism is transparent for a user but should be taken into account when debugging.

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.

2.7.3 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.

2.7.4 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 7, and this causes problems for simple windows (objects of `java.awt.Window`class). 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.

2.7.5 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: MS Windows. A component requests focus and is concurrently removed from its container.

Sometimes `java.lang.NullPointerException: null pData` is thrown.

Solution: The easiest way to avoid throwing the exception is to do the removal along with requesting focus on EDT. Another, more complicated, approach is to synchronize requesting focus and removal if you need to perform these actions on different threads.

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 focus owner.

For example, a frame is the focused window, and one of its components is the focus owner. Another window is clicked, and then the frame is clicked again. `WINDOW_GAINED_FOCUS` comes to the frame and the `WindowFocusListener.windowGainedFocus(WindowEvent)` method is called. However, inside of this callback you cannot determine the frame's most recent focus owner, because `Frame.getMostRecentFocusOwner()` returns null.

Solution: You can get the frame's most recent focus owner inside the `WindowListener.windowActivated(WindowEvent)` callback. However, by this time the frame will have become the focused window only if it does not have owned windows. Note 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
- Swing guide

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("SomeTransferedTe
             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 7 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.

2.9.6 Hangs when debugging popup menus and similar components on X11

Certain GUI actions require grabbing all the input events in order to determine when the action should terminate (e.g. navigating popup menus). While the grab is active, no other applications receive input events. If a Java application is being debugged, and reached a breakpoint while the grab has been active, the operating system appears hanged. This happens because the Java application holding the grab is stopped by the debugger and can't process any input events, and other applications simply don't receive the events due to the installed grab. In order to allow debugging such applications, the following system property should be set when running the application from the debugger:

```
-Dsun.awt.disablegrab=true
```

This effectively turns off setting the grab, and as such doesn't hang the system. However, with this option set, in some cases this may lead to inability to terminate a GUI actions that would normally be terminated. For example, popup menus may not be dismissed when clicking window's titlebar.

2.9.7 `Window.toFront()/toBack()` behavior on X11

Due to restrictions enforced by third-party software (in particular, by window managers such as the Metacity), the `toFront()/toBack()` methods may not always work as expected and cause the window to not change its stacking order in relation to other top-level windows. More details are available in the CR 6472274.

If an application ultimately wants to bring a window to top, it can try to workaround the issue by temporarily making the window "always on top" calling the `Window.setAlwaysOnTop(true)` and then calling `setAlwaysOnTop(false)` to reset the "always on top" state. Note that this workaround isn't guaranteed to work either because window managers can enforce more restrictions in the future. Also please note that setting a window "always on top" is available to trusted applications only. An unsigned applet or an unsigned web-start application running in a sandbox can't use this API, and thus is unable to workaround the issue.

However, native applications experience similar issues, and as such this peculiarity makes Java applications behave similar to native applications. Therefore this issue cannot be considered as a bug.

2.10 Heavyweight/Lightweight Components Mixing

This section discusses possible issues with the heavyweight/lightweight (HW/LW) mixing feature.

2.10.1 The requirement of validating the component hierarchy

Changing any layout-related properties of a component, such as its size, location, or font, invalidates the component as well as its ancestors. In order for the HW/LW Mixing feature to function correctly, the component hierarchy must be validated after making such changes. By default, invalidation stops on the top-most container of the hierarchy (for example, a `Frame` object). Therefore, to restore the validity of the hierarchy the application should call the `Frame.validate()` method. For example:

```
component.setFont(myFont);
frame.validate();
```

where `frame` refers to a frame which contains `component`. Note that Swing applications and the Swing library itself often use the following pattern:

```
component.setFont(myFont);
component.revalidate();
```

The `revalidate()` call is **not** sufficient because it validates the hierarchy starting from the nearest validate root of the component only, thus leaving the upper containers invalid. In that case, the HW/LW feature may not calculate correct shapes for heavyweight components, and visual artifacts may be seen on the screen.

To verify the validity of the whole component hierarchy a user can use the key combination Ctrl+Shift+F1 as described in 2.1 of this document. A component marked 'invalid' may indicate a missing `validate()` call somewhere.

2.10.2 Validate roots

The concept of validate roots mentioned in 2.10.1 has been introduced in Swing in order to speed up the process of validating component hierarchies because it may take a significant amount of time. While such optimization leaves upper parts of hierarchies invalid, this didn't bring any issues because the layout of components inside a validate root doesn't affect the layout of outside component hierarchy (that is, the siblings of the validate root). However, when HW and LW components are mixed together in a hierarchy, this statement is no longer true. That's why the feature requires the whole component hierarchy to be valid.

Calling `frame.validate()` may be inefficient as well, and as such AWT supports an alternative, optimized way of handling invalidation/validation of component hierarchies. This feature is enabled with a system property:

```
-Djava.awt.smartInvalidate=true
```

Once this property is specified, the `invalidate()` method will stop invalidation of the hierarchy when it reaches the nearest validate root of a component the `invalidate()` method has been invoked on. Afterwards the application should simply call:

```
component.revalidate();
```

to restore the validity of the component hierarchy. Note that in this case calling `frame.validate ()` would be effectively a no-op because frame is still valid. Since some applications rely on calling `validate()` directly on a component upper than the validate root of the hierarchy (for example, a frame), this new optimized behavior may cause incompatibility issues, and hence it's available only when specifying the system property.

If an application experiences any difficulties running in this new optimized mode, a user can use the key combination Ctrl+Shift+F1 as described in 2.1 of this document to investigate what parts of the component hierarchy are left invalid, and thus possibly cause the problems.

2.10.3 Swing painting optimizations

By default, Swing library assumes that there are no heavyweight components in the component hierarchy, and therefore uses optimized drawing techniques to boost performance of the Swing UI. If a component hierarchy contains hw components, the optimizations must be turned off. This is relevant for Swing `JScrollPane`s in the first place. You can change the scrolling mode by using the `JViewPort.setScrollMode(int)` method.

2.10.4 Non-opaque lightweight components

Non-opaque lightweight components are not supported by the hw/lw mixing feature implementation by default. In order to enable mixing non-rectangular lw components with hw components, the application must use the `com.sun.awt.AWTUtilities.setComponentMixingCutoutShape()` non-public API.

Note that non-rectangular lw components should still paint themselves using either opaque (alpha == 1.0) or transparent (alpha == 0.0) colors. Using translucent colors (with 0.0 < alpha < 1.0) is not supported.

2.10.5 Disabling the default hw/lw mixing feature implementation

In the past, some developers have implemented their own support for cases when hw and lw components must be mixed together. The built-in implementation of the feature available since JDK 6 update 12 and JDK 7 may cause problems with custom workarounds. In order to disable the built-in feature the application must be started with the following system property:

```
-Dsun.awt.disableMixing=true
```