# AWT Threading Issues

## Listeners and threads

Unless otherwise noted all AWT listeners are notified on the event dispatch thread. It is safe to remove/add listeners from any thread during dispatching, but the changes only effect subsequent notification.

For example, if a key listeners is added from another key listener, the newly added listener is only notified on subsequent key events.

## Auto-shutdown

According to [*The Java*TM *Virtual Machine Specification, Second edition*](http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html) (see [�2.17.9](http://java.sun.com/docs/books/vmspec/2nd-edition/html/Concepts.doc.html#19152) and [�2.19](http://java.sun.com/docs/books/vmspec/2nd-edition/html/Concepts.doc.html#33308)), the Java virtual machine (JVM) initially starts up with a single non-daemon thread, which typically calls the main method of some class. The virtual machine terminates all its activity and exits when one of two things happens:

* All the threads that are not daemon threads terminate.
* Some thread invokes the exit method of class Runtime or class System, and the exit operation is permitted by the security manager.

This implies that if an application doesn't start any threads itself, the JVM will exit as soon as main terminates. This is not the case, however, for a simple application that creates and displays a java.awt.Frame:

public static void main(String[] args) {  
 Frame frame = new Frame();  
 frame.setVisible(true);  
 }

The reason is that AWT encapsulates asynchronous event dispatch machinery to process events AWT or Swing components can fire. The exact behavior of this machinery is implementation-dependent. In particular, it can start non-daemon helper threads for its internal purposes. In fact, these are the threads that prevent the example above from exiting. The only restrictions imposed on the behavior of this machinery are as follows:

* [EventQueue.isDispatchThread](http://docs.google.com/EventQueue.html#isDispatchThread()) returns true if and only if the calling thread is the event dispatch thread started by the machinery;
* AWTEvents which were actually enqueued to a particular EventQueue (note that events being posted to the EventQueue can be coalesced) are dispatched:
  + Sequentially. That is, it is not permitted that several events from this queue are dispatched simultaneously.
    - In the same order as they are enqueued. That is, if AWTEvent A is enqueued to the EventQueue before AWTEvent B then event B will not be dispatched before event A.
* There is at least one alive non-daemon thread while there is at least one displayable AWT or Swing component within the application (see [Component.isDisplayable](http://docs.google.com/Component.html#isDisplayable())).

The implications of the third restriction are as follows:

* The JVM will exit if some thread invokes the exit method of class Runtime or class System regardless of the presence of displayable components;
* Even if the application terminates all non-daemon threads it started, the JVM will not exit while there is at least one displayable component.

It depends on the implementation if and when the non-daemon helper threads are terminated once all components are made undisplayable. The implementation-specific details are given below.

### Implementation-dependent behavior.

Prior to 1.4, the helper threads were never terminated.

Starting with 1.4, the behavior has changed as a result of the fix for  [4030718](http://developer.java.sun.com/developer/bugParade/bugs/4030718.html). With the current implementation, AWT terminates all its helper threads allowing the application to exit cleanly when the following three conditions are true:

* There are no displayable AWT or Swing components.
* There are no native events in the native event queue.
* There are no AWT events in java EventQueues.

Therefore, a stand-alone AWT application that wishes to exit cleanly without calling System.exit must:

* Make sure that all AWT or Swing components are made undisplayable when the application finishes. This can be done by calling [Window.dispose](http://docs.google.com/Window.html#dispose()) on all top-level Windows. See [Frame.getFrames](http://docs.google.com/Frame.html#getFrames()).
* Make sure that no method of AWT event listeners registered by the application with any AWT or Swing component can run into an infinite loop or hang indefinitely. For example, an AWT listener method triggered by some AWT event can post a new AWT event of the same type to the EventQueue. The argument is that methods of AWT event listeners are typically executed on helper threads.

Note, that while an application following these recommendations will exit cleanly under normal conditions, it is not guaranteed that it will exit cleanly in all cases. Two examples:

* Other packages can create displayable components for internal needs and never make them undisplayable. See  [4515058](http://developer.java.sun.com/developer/bugParade/bugs/4515058.html),  [4671025](http://developer.java.sun.com/developer/bugParade/bugs/4671025.html), and  [4465537](http://developer.java.sun.com/developer/bugParade/bugs/4465537.html).
* Both Microsoft Windows and X11 allow an application to send native events to windows that belong to another application. With this feature it is possible to write a malicious program that will continuously send events to all available windows preventing any AWT application from exiting cleanly.

On the other hand, if you require the JVM to continue running even after the application has made all components undisplayable you should start a non-daemon thread that blocks forever.

<...>  
 Runnable r = new Runnable() {  
 public void run() {  
 Object o = new Object();  
 try {  
 synchronized (o) {  
 o.wait();  
 }  
 } catch (InterruptedException ie) {  
 }  
 }  
 };  
 Thread t = new Thread(r);  
 t.setDaemon(false);  
 t.start();  
 <...>

The Java Virtual Machine Specification guarantees that the JVM doesn't exit until this thread terminates.