# Why Are Thread.stop, Thread.suspend,  Thread.resume and Runtime.runFinalizersOnExit Deprecated?

<https://docs.oracle.com/javase/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html>

### Why is Thread.stop deprecated?

Because it is inherently unsafe. Stopping a thread causes it to unlock all the monitors that it has locked. (The monitors are unlocked as the ThreadDeath exception propagates up the stack.) If any of the objects previously protected by these monitors were in an inconsistent state, other threads may now view these objects in an inconsistent state. Such objects are said to be *damaged*. When threads operate on damaged objects, arbitrary behavior can result. This behavior may be subtle and difficult to detect, or it may be pronounced. Unlike other unchecked exceptions, ThreadDeath kills threads silently; thus, the user has no warning that his program may be corrupted. The corruption can manifest itself at any time after the actual damage occurs, even hours or days in the future.

### Couldn't I just catch the ThreadDeath exception and fix the damaged object?

In theory, perhaps, but it would vastly complicate the task of writing correct multithreaded code. The task would be nearly insurmountable for two reasons:

1. A thread can throw a ThreadDeath exception *almost anywhere*. All synchronized methods and blocks would have to be studied in great detail, with this in mind.
2. A thread can throw a second ThreadDeath exception while cleaning up from the first (in the catch or finally clause). Cleanup would have to repeated till it succeeded. The code to ensure this would be quite complex.

In sum, it just isn't practical.

### What about Thread.stop(Throwable)?

In addition to all of the problems noted above, this method may be used to generate exceptions that its target thread is unprepared to handle (including checked exceptions that the thread could not possibly throw, were it not for this method). For example, the following method is behaviorally identical to Java's throw operation, but circumvents the compiler's attempts to guarantee that the calling method has declared all of the checked exceptions that it may throw:

static void sneakyThrow(Throwable t) {

Thread.currentThread().stop(t);

}

### What should I use instead of Thread.stop?

Most uses of stop should be replaced by code that simply modifies some variable to indicate that the target thread should stop running. The target thread should check this variable regularly, and return from its run method in an orderly fashion if the variable indicates that it is to stop running. (This is the approach that the Java Tutorial has always recommended.) To ensure prompt communication of the stop-request, the variable must be volatile (or access to the variable must be synchronized).

For example, suppose your applet contains the following start, stop and run methods:

private Thread blinker;

public void start() {

blinker = new Thread(this);

blinker.start();

}

public void stop() {

blinker.stop(); // UNSAFE!

}

public void run() {

Thread thisThread = Thread.currentThread();

while (true) {

try {

thisThread.sleep(interval);

} catch (InterruptedException e){

}

repaint();

}

}

You can avoid the use of Thread.stop by replacing the applet's stop and run methods with:

private volatile Thread blinker;

public void stop() {

blinker = null;

}

public void run() {

Thread thisThread = Thread.currentThread();

while (blinker == thisThread) {

try {

thisThread.sleep(interval);

} catch (InterruptedException e){

}

repaint();

}

}

### How do I stop a thread that waits for long periods (e.g., for input)?

That's what the Thread.interrupt method is for. The same "state based" signaling mechanism shown above can be used, but the state change (blinker = null, in the previous example) can be followed by a call to Thread.interrupt, to interrupt the wait:

public void stop() {

Thread moribund = waiter;

waiter = null;

moribund.interrupt();

}

For this technique to work, it's critical that any method that catches an interrupt exception and is not prepared to deal with it immediately reasserts the exception. We say reasserts rather than rethrows, because it is not always possible to rethrow the exception. If the method that catches the InterruptedException is not declared to throw this (checked) exception, then it should "reinterrupt itself" with the following incantation:

Thread.currentThread().interrupt();

This ensures that the Thread will reraise the InterruptedException as soon as it is able.

### What if a thread doesn't respond to Thread.interrupt?

In some cases, you can use application specific tricks. For example, if a thread is waiting on a known socket, you can close the socket to cause the thread to return immediately. Unfortunately, there really isn't any technique that works in general. It should be noted that in all situations where a waiting thread doesn't respond to Thread.interrupt, it wouldn't respond to Thread.stop either.Such cases include deliberate denial-of-service attacks, and I/O operations for which thread.stop and thread.interrupt do not work properly.

### Why are Thread.suspend and Thread.resume deprecated?

Thread.suspend is inherently deadlock-prone. If the target thread holds a lock on the monitor protecting a critical system resource when it is suspended, no thread can access this resource until the target thread is resumed. If the thread that would resume the target thread attempts to lock this monitor prior to calling resume, deadlock results. Such deadlocks typically manifest themselves as "frozen" processes.

### What should I use instead of Thread.suspend and Thread.resume?

As with Thread.stop, the prudent approach is to have the "target thread" poll a variable indicating the desired state of the thread (active or suspended). When the desired state is suspended, the thread waits using Object.wait. When the thread is resumed, the target thread is notified using Object.notify.

For example, suppose your applet contains the following mousePressed event handler, which toggles the state of a thread called blinker:

private boolean threadSuspended;

Public void mousePressed(MouseEvent e) {

e.consume();

if (threadSuspended)

blinker.resume();

else

blinker.suspend(); // DEADLOCK-PRONE!

threadSuspended = !threadSuspended;

}

You can avoid the use of Thread.suspend and Thread.resume by replacing the event handler above with:

public synchronized void mousePressed(MouseEvent e) {

e.consume();

threadSuspended = !threadSuspended;

if (!threadSuspended)

notify();

}

and adding the following code to the "run loop":

synchronized(this) {

while (threadSuspended)

wait();

}

The wait method throws the InterruptedException, so it must be inside a try ... catch clause. It's fine to put it in the same clause as the sleep. The check should follow (rather than precede) the sleep so the window is immediately repainted when the the thread is "resumed." The resulting run method follows:

public void run() {

while (true) {

try {

Thread.currentThread().sleep(interval);

synchronized(this) {

while (threadSuspended)

wait();

}

} catch (InterruptedException e){

}

repaint();

}

}

Note that the notify in the mousePressed method and the wait in the run method are inside synchronized blocks. This is required by the language, and ensures that wait and notify are properly serialized. In practical terms, this eliminates race conditions that could cause the "suspended" thread to miss a notify and remain suspended indefinitely.

While the cost of synchronization in Java is decreasing as the platform matures, it will never be free. A simple trick can be used to remove the synchronization that we've added to each iteration of the "run loop." The synchronized block that was added is replaced by a slightly more complex piece of code that enters a synchronized block only if the thread has actually been suspended:

if (threadSuspended) {

synchronized(this) {

while (threadSuspended)

wait();

}

}

In the absence of explicit synchronization, threadSuspended must be made volatile to ensure prompt communication of the suspend-request.

The resulting run method is:

private boolean volatile threadSuspended;

public void run() {

while (true) {

try {

Thread.currentThread().sleep(interval);

if (threadSuspended) {

synchronized(this) {

while (threadSuspended)

wait();

}

}

} catch (InterruptedException e){

}

repaint();

}

}

### Can I combine the two techniques to produce a thread that may be safely "stopped" or "suspended"?

Yes; it's reasonably straightforward. The one subtlety is that the target thread may already be suspended at the time that another thread tries to stop it. If the stop method merely sets the state variable (blinker) to null, the target thread will remain suspended (waiting on the monitor), rather than exiting gracefully as it should. If the applet is restarted, multiple threads could end up waiting on the monitor at the same time, resulting in erratic behavior.

To rectify this situation, the stop method must ensure that the target thread resumes immediately if it is suspended. Once the target thread resumes, it must recognize immediately that it has been stopped, and exit gracefully. Here's how the resulting runand stop methods look:

public void run() {

Thread thisThread = Thread.currentThread();

while (blinker == thisThread) {

try {

thisThread.sleep(interval);

synchronized(this) {

while (threadSuspended && blinker==thisThread)

wait();

}

} catch (InterruptedException e){

}

repaint();

}

}

public synchronized void stop() {

blinker = null;

notify();

}

If the stop method calls Thread.interrupt, as described above, it needn't call notify as well, but it still must be synchronized. This ensures that the target thread won't miss an interrupt due to a race condition.

### What about Thread.destroy?

Thread.destroy has never been implemented. If it were implemented, it would be deadlock-prone in the manner of Thread.suspend. (In fact, it is roughly equivalent to Thread.suspend without the possibility of a subsequent Thread.resume.) We are not implementing it at this time, but neither are we deprecating it (forestalling its implementation in future). While it would certainly be deadlock prone, it has been argued that there may be circumstances where a program is willing to risk a deadlock rather than exit outright.

### Why is Runtime.runFinalizersOnExit deprecated?

Because it is inherently unsafe. It may result in finalizers being called on live objects while other threads are concurrently manipulating those objects, resulting in erratic behavior or deadlock. While this problem could be prevented if the class whose objects are being finalized were coded to "defend against" this call, most programmers do *not* defend against it. They assume that an object is dead at the time that its finalizer is called.

Further, the call is not "thread-safe" in the sense that it sets a VM-global flag. This forces *every* class with a finalizer to defend against the finalization of live objects!