
**Technical Document Series**

# POSIX Signal Handling in Java

# POSIX Signal Handling In Java

## Introduction

POSIX signals inform a running process of external events, such as the user wishing to kill the process, or the operating system signaling an impending shutdown, or the process being suspended or reinstated; or the process may have violated a resource constraint, such as excessive CPU usage or attempts to access areas outside its permitted memory space, and is asked to shutdown. In short, POSIX signals serve many different purposes. Some are even up to interpretation, such as the HUP (HangUP) signal, which is commonly used to inform a process that something about its environment has changed and the process should adjust accordingly. Some programs may interpret this to mean that the configuration has changed and needs to be reloaded; or the log file has been moved for archiving purposes and a new one should be started. The use of signals is widespread, especially on Unix-based operating systems, but Java provides no standard interface for a Java application to hear and react to them.

This document shows you how to get around this limitation.

## The Good, the Bad, and the Ugly

The good news is that there is a way to intercept POSIX signals and react to them in Java. This would allow your Java program to avoid being killable with ^C (SIGINT), for example, even ignore termination requests from the operating system (SIGTERM). Neither of these is necessarily a good idea, of course, unless you know exactly why you would want to catch these signals and either handle them yourself or ignore them altogether. Suffice to say that catching signals such as HUP and USR2 could allow your Java software to play by rules that many non-Java programs follow as a matter of course, thereby integrating them more fully into your overall software suite.

The bad news is that there is no *standard interface* for Java applications to handle POSIX signals. The only one known to this author at this time is implemented in the `sun.misc` package which is, of course, a proprietary package controlled by Sun Microsystems. Sun could change or even remove this package at any time, so our examples (and, by extension, your code) would build on a strong dependency on Sun's libraries. This means that our code will not work with IBM's Java, or the Blackdown Java packages, or any other vendor's libraries. You may find that these vendors have implementations for signal handling that are similar to Sun's. You may be able to adapt this code to those implementations, but you won't be able to run our code “out of the box” with another vendor's libraries.

Here is hoping that Sun provides access to POSIX signals in a future release!

## Signals Used by the JVM

Of course, the JVM uses signals. Some implementations—and we're not just talking about Sun's JVM—may be rely on catching SIGINT or SIGTERM to run the various shutdown hooks in your code before actually terminating. If your code caught and ignored SIGTERM, for example, the operating system might not be able to terminate your software as part of shutdown operations, either causing the shutdown to stretch in time, or your software would get killed without having the chance to shutdown gracefully. Some signals, such as HUP, are widely used to notify processes of some expected event which makes them fairly safe to use, but not all signals should be meddled with.

If you are unable to listen to a particular signal then it is either because the signal is not supported on your operating system (Microsoft Windows does not appear to support the HUP signal, DUH!) or the JVM is already listening to the signal. Signals can only go to a single interested party, not two. If you cannot use a different signal then you must ask the JVM to reduce its reliance on signals. The Sun Microsystems JVM accepts the -Xrs flag for this purpose. Other vendors might provide a similar flag. If not, you may be out of luck.

## The Code

The source on the following two pages separates the dependent from the independent code. We make no claim that this is the proper way to handle signals, nor is it likely the most efficient, but it works and that is ultimately the most important factor.

The first portion has no dependencies on Sun Microsystems libraries at all. That is the code that you should study most carefully as you would want to implement in your own code. Most succinctly, you want to create a class that implements the `Observer` interface, supplying the `update` method that is called whenever a signal is received. You'd then create the `SignalHandler` instance (the second portion of our source code) and register the `Observer` with it. Once you've done that, ask the `SignalHandler` to handle all the signals you want to hear about, using the `handleSignal` method.

Our example program does this and then waits for ten seconds before terminating. During that time you can find out the process ID and send it signals with the kill command. The example below shows two windows, with the software having run in one, and the kills having been sent in the other.

```
% javac SignalTest.java
% java SignalTest
Sleeping for 10 seconds; hit me with signals!
Received signal: SIGHUP
Received signal: SIGHUP
% _
```

```
% ps aux | grep java | grep SignalTest
udo      1802  0.0  0.1 .........
% kill -HUP 1802
% kill -HUP 1802
% _
```

## Subtle Caveats

When this code runs on any JVM other than Sun Microsystems' (or, more broadly, on a JVM that does not implement the specific classes in the same package, with the same API as what we're coding against) then a variety of exceptions and/or errors will be thrown by any attempt to instantiate the `SignalHandler` class. We guard against this fairly generically by simply catching `Throwable` which is the superclass of all `Error` and `Exception` classes.

 

```
import java.util.Observer;
import java.util.Observable;

/**
 * <p>SignalTest serves two purposes:</p>
 *
 * <ol>
 * <li>It creates and sets up the signal handling (in the constructor)
 * <li>It implements Observer and provides the update method that gets
 *     called whenever a signal is received
 * </ol>
 *
 * <p>Merely instantiating this object (and hanging onto a reference
 * to it) is enough to enable the signal handling.</p>
 **/
public class SignalTest
  implements Observer
{
  /**
   * The software's entry point; exits after 10 seconds.
   * @param args Any and all arguments are ignored and have no effect.
   **/
  public static void main( final String[] args )
  {
    new SignalTest().go();
  }

  private void go()
  {
    try
      {
        final SignalHandler sh = new SignalHandler();
        sh.addObserver( this );
        sh.handleSignal( "HUP" );

        System.out.println( "Sleeping for 10 seconds: hit me with signals!" );
        Thread.sleep( 10000 );
      }
    catch( Throwable x )
      {
        // SignalHandler failed to instantiate: maybe the classes do not exist,
        // or the API has changed, or something else went wrong; actualy we get
        // here on an InterruptedException from Thread.sleep, too, but that is
        // probably quite rare and doesn't matter in a simple demo like this.
        x.printStackTrace();
      }
  }

  /**
   * Implementation of Observer, called by {@link SignalHandler} when
   * a signal is received.
   * @param o Our {@link SignalHandler} object
   * @param arg The {@link sun.misc.Signal} that triggered the call
   **/
  public void update( final Observable o,
                      final Object arg )
  {
    // use the same method that the Timer employs to trigger a
    // rotation, which ensures that signal and timer don't screw
    // each other up.
    System.out.println( "Received signal: "+arg );
  }
}
```

```
/**
 * <p>An implementation of Sun Microsystems' {@link
 * sun.misc.SignalHandler} interface that is also {@link Observable} so
 * that we can notify {@link Observer}S when a signal is raised. The
 * {@link #handle(sun.misc.signal)} method is called by Sun's libraries
 * for every signal received that was registered with a call to Sun's
 * static {@link sun.misc.Signal#handle(sun.misc.Signal,sun.misc.SignalHandle)}
 * method.</p>
 **/
class SignalHandler
  extends Observable
  implements sun.misc.SignalHandler
{
  /**
   * Tells the object to handle the given signal.
   * @param signalName The name of the signal, such as "SEGV", "ILL",
   * "FPE", "ABRT", "INT", "TERM", "HUP", etc. Not all platforms
   * support all signals. Microsoft Windows may not support HUP, for
   * example, whereas that is a widely use and supported signal under
   * Unix (and its variants); additionally, the JVM may be using some
   * signals (the use of -Xrs will reduce or disable them at the cost
   * of losing what the JVM wanted them for).
   * @exception IllegalArgumentException is thrown when the named
   * signal is not available for some reason. Watch out: the original
   * cause (missing class or method) may be wrapped inside the exception!
   **/
  public void handleSignal( final String signalName )
    throws IllegalArgumentException
  {
    try
      {
        sun.misc.Signal.handle( new sun.misc.Signal(signalName), this );
      }
    catch( IllegalArgumentException x )
      {
        // Most likely this is a signal that's not supported on this
        // platform or with the JVM as it is currently configured
        throw x;
      }
    catch( Throwable x )
      {
        // We may have a serious problem, including missing classes
        // or changed APIs
        throw new IllegalArgumentException( "Signal unsupported: "+signalName, x );
      }
  }

  /**
   * Called by Sun Microsystems' signal trapping routines in the JVM.
   * @param signal The {@link sun.misc.Signal} that we received
   **/
  public void handle( final sun.misc.Signal signal )
  {
    // setChanged ensures that notifyObservers actually calls someone. In
    // simple cases this seems like extra work but in asynchronous designs,
    // setChanged might be called on one thread, and notifyObservers, on
    // another or only when multiple changes may have been completed (to
    // wrap up multiple changes in a single notifcation).
    setChanged();
    notifyObservers( signal );
  }
}
```

## Bonus Section

The following describes what happens when a signal is not blocked *and* set to a default behavior (which is not how most software configures itself, to be sure). You can largely ignore the second portion of each column. See `<signal.c>` for more details:

| | | | | | |
|---|---|---|---|---|---|
| SIGHUP | terminate | SIGUSR2 | terminate | SIGXCPU | coredump |
| SIGINT | terminate | SIGPIPE | terminate | SIGXFSZ | coredump |
| SIGQUIT | coredump | SIGALRM | terminate | SIGVTALRM | terminate |
| SIGILL | coredump | SIGTERM | terminate | SIGPROF | terminate |
| SIGTRAP | coredump | SIGCHLD | ignore | SIGPOLL/SIGIO | terminate |
| SIGABRT/SIGIOT | coredump | SIGCONT | ignore | SIGSYS/SIGUNUSED | coredump |
| SIGBUS | coredump | SIGSTOP | stop | SIGSTKFLT | terminate |
| SIGFPE | coredump | SIGTSTP | stop | SIGWINCH | ignore |
| SIGKILL | terminate | SIGTTIN | stop | SIGPWR | terminate |
| SIGUSR1 | terminate | SIGTTOU | stop | SIGRTMIN-SIGRTMAX | terminate |
| SIGSEGV | coredump | SIGURG | ignore | | |

## Summary

Keep the following points in mind as you use signals in Java, whether or not you base your solution on the code above, or another variation:

- As of Java 1.5 no published (and therefore standard) classes exist to intercept and handle signals,
- Implementations such as demonstrated above will vary from vendor to vendor, and may even change from one release of the Java libraries to the next,
- Not all signals are supported on all platforms that can run Java; the JVM for Microsoft Windows, for example, does not seem to support the SIGHUP used in the demonstration code above and Windows does not even seem to support the raising of signals as part of normal procedures: We used cygwin for a Unix-like environment but are still constrained by what Microsoft supports and does not.
- Our source code formatting may diverge from recommended and accepted standards but we have our reasons.