From: **Erhard Plödereder** ploedere@iste.uni-stuttgart.de
Subject: Re: FibonacciTaskUsingInterrupt
Date: June 24, 2026 at 11:03 PM
To: McDonagh, Sean Sean.McDonagh@us.amentum.com, Stephen Michell stephen.michell@maurya.on.ca, tullio.vardanega@math.unipd.it, Larry Wagoner ldw11@aol.com

EP

After reading Sean's mails on the subject, I have second thoughts about our advice, since the one about termination via flag indeed ignores the situation of deadlocked threads as a reason for termination from outside and doesn't mention that limitation.

The problem for the advice section is that, with interrupt, one had to advise that there is both a synchronous isInterrupted check and a handling of the exception. The former is mandatory and the latter equally so unless it is guaranteed that the thread does not block/wait. So, what is better? One mechanism (interrupt) with a relatively complicated application rule? Or two mechanisms, one for terminating a non-blocking thread, and another one for potentially blocking threads?
I am gradually moving over to the "use interrupt only" camp with the advice to always have a synchronous isInterrupted check **and** a handling of the exception. (and leave its omission for nonblocking threads to the description, possibly with the rationale that the blocking might not be immediately knowable,e.g., it might happen in subcalls.)

So, let's please talk about it again.

Erhard

Am 24.06.2026 um 21:35 schrieb McDonagh, Sean:

> **1. The Volatile Flag Approach (Flawed for Blocking Tasks)**
> - **How it works:** The worker thread regularly checks a boolean variable (or status object) on every loop iteration to see if it should exit.
> - **Why it fails during blocking:** If the thread is executing a blocking operation—such as Thread.sleep(), Object.wait(), or reading from an un-buffered network socket—it is entirely paused. It cannot execute the loop's conditional check.
> - **The consequence:** The thread remains completely frozen and unresponsive to the shutdown request until the blocking operation naturally finishes or times out.
>
> **2. The Thread.interrupt() Approach (Correct Native Standard)**
> - **How it works:** Calling interrupt() does not forcefully kill a thread. Instead, it alters an internal JVM flag status inside the targeted thread's metadata.
> - **Why it succeeds during blocking:** Built-in Java blocking methods are explicitly engineered to continuously monitor this internal JVM status flag. The moment the flag is set, the blocking method immediately wakes up the thread, clears the block, and throws an InterruptedException.
> - **The consequence:** The thread can instantly enter its catch block to clean up resources and exit, completely eliminating shutdown delays.
>
> ---
>
> ```
> public class FibonacciThreadTest {
>
>     public static void main(String[] args) throws InterruptedException {
>         System.out.println("=== STARTING CONCURRENT SHUTDOWN TEST ===");
>
>         // ------------------------------------------------------------
>         // TEST 1: The Flag Approach (Flawed)
>         // ------------------------------------------------------------
>         long startFlag = System.currentTimeMillis();
>         System.out.println("\n[0ms] Main: Starting flag-based task...");
>         FibonacciTaskUsingFlag flagTask = new FibonacciTaskUsingFlag();
>         flagTask.start();
>
>         // Let the task run briefly (it immediately enters a 4-second sleep)
>         Thread.sleep(1500);
>         System.out.println("[" + (System.currentTimeMillis() - startFlag) + "ms] Main:
> Signaling flag-based task to stop...");
> ```

```
Signaling flag-based task to stop... ");
        flagTask.triggerExit();

        // Block main thread until the flag thread actually finishes dying
        flagTask.join();
        System.out.println("[" + (System.currentTimeMillis() - startFlag) + "ms] Main: Flag-
based task has completely exited.");

        // ------------------------------------------------------------
        // TEST 2: The Interrupt Approach (Correct)
        // ------------------------------------------------------------
        long startInterrupt = System.currentTimeMillis();
        System.out.println("\n[0ms] Main: Starting interrupt-based task...");
        FibonacciTaskUsingInterrupt interruptTask = new FibonacciTaskUsingInterrupt();
        interruptTask.start();

        // Let the task run briefly (it immediately enters a 4-second sleep)
        Thread.sleep(1500);
        System.out.println("[" + (System.currentTimeMillis() - startInterrupt) + "ms] Main:
Signaling interrupt-based task to stop...");
        interruptTask.interrupt();

        // Block main thread until the interrupt thread finishes dying
        interruptTask.join();
        System.out.println("[" + (System.currentTimeMillis() - startInterrupt) + "ms] Main:
Interrupt-based task has completely exited.");
    }

    // --- APPROACH 1: FLAWED FLAG DESIGN ---
    static class FibonacciTaskUsingFlag extends Thread {
        private volatile boolean exitFlag = false;

        public void triggerExit() {
            this.exitFlag = true;
        }

        @Override
        public void run() {
            int n1 = 0, n2 = 1;
            for (int i = 0; i < 5; i++) {
                if (exitFlag) {
                    System.out.println("  -> [TaskFlag] Flag check caught! Loop exiting.");
                    return;
                }

                try {
                    // Simulating a blocking operation or time delay
                    Thread.sleep(4000);
                } catch (InterruptedException e) {
                    // Left empty because flag architecture doesn't focus on interrupts
                }

                int sum = n1 + n2;
                n1 = n2;
```

```
            n2 = sum;
            System.out.println("  -> [TaskFlag] Calculated Fib sequence value: " + sum);
        }
    }
}

// --- APPROACH 2: CORRECT INTERRUPT DESIGN ---
static class FibonacciTaskUsingInterrupt extends Thread {
    @Override
    public void run() {
        int n1 = 0, n2 = 1;
        for (int i = 0; i < 5; i++) {
            // Good practice to explicitly poll the status if performing heavy raw math
            if (Thread.currentThread().isInterrupted()) {
                System.out.println("  -> [TaskInterrupt] Interrupt status caught via poll!
Exiting.");
                return;
            }

            try {
                // Native JVM blocking operation responds directly to interrupt()
                Thread.sleep(4000);
            } catch (InterruptedException e) {
                System.out.println("  -> [TaskInterrupt] InterruptedException thrown! Sleep
broken instantly.");
                // Restore interrupt status flag and exit
                Thread.currentThread().interrupt();
                return;
            }

            int sum = n1 + n2;
            n1 = n2;
            n2 = sum;
            System.out.println("  -> [TaskInterrupt] Calculated Fib sequence value: " + sum);
        }
    }
}
}
```

From: McDonagh, Sean <Sean.McDonagh@us.amentum.com>
Sent: Wednesday, June 24, 2026 2:57 PM
To: Stephen Michell <stephen.michell@maurya.on.ca>; Erhard Plödereder <ploedere@iste.uni-stuttgart.de>; tullio.vardanega@math.unipd.it; Larry Wagoner <ldw11@aol.com>
Subject: Runnable Example

This one example shows both bad and good and runs consistently for me ….

```
public class ThreadCancellationDemo {

    // Custom flag used in the broken example
    private static volatile boolean keepRunning = true;
```

```
    public static void main(String[] args) throws InterruptedException {
        System.out.println("--- SCENARIO 1: THE BROKEN CUSTOM FLAG ---");
        runBrokenFlagScenario();

        System.out.println("\n--- SCENARIO 2: THE FIXED NATIVE INTERRUPT ---");
        runFixedInterruptScenario();
    }

    /**
     * SCENARIO 1: This thread will FAIL to stop.
     * It gets stuck inside the 10-second sleep and ignores the flag change.
     */
    private static void runBrokenFlagScenario() throws InterruptedException {
        Thread flagWorker = new Thread(() -> {
            while (keepRunning) {
                try {
                    System.out.println("  [FlagWorker] Starting a long 10-second sleep...");
                    Thread.sleep(10000);
                    System.out.println("  [FlagWorker] Woke up naturally.");
                } catch (InterruptedException e) {
                    System.out.println("  [FlagWorker] This will never happen because we don't
interrupt.");
                }
            }
            System.out.println("  [FlagWorker] Thread stopped cleanly.");
        });

        flagWorker.start();
        Thread.sleep(1000); // Let the thread enter its 10-second sleep

        System.out.println("[Main] Setting keepRunning flag to false...");
        keepRunning = false; // Changing the flag does absolutely nothing to wake it up!

        System.out.println("[Main] Waiting 2 seconds to see if FlagWorker dies...");
        flagWorker.join(2000);

        if (flagWorker.isAlive()) {
            System.out.println("[Main] ❌ FAILURE: FlagWorker is still alive and trapped in
sleep!");
        }
    }

    /**
     * SCENARIO 2: This thread SUCCEEDS in stopping.
     * The native interrupt instantly wakes it from sleep, bypassing the remaining 10
seconds.
     */
    private static void runFixedInterruptScenario() throws InterruptedException {
        Thread interruptWorker = new Thread(() -> {
            // FIX Part A: Check the native interrupt status instead of a custom flag
            while (!Thread.currentThread().isInterrupted()) {
                try {
                    System.out.println("  [InterruptWorker] Starting a long 10-second sleep...");
```

```
                Thread.sleep(10000);
                System.out.println("  [InterruptWorker] Woke up naturally.");
            } catch (InterruptedException e) {
                System.out.println("  [InterruptWorker] 💥 Caught InterruptedException!
Woke up instantly.");

                // FIX Part B: Re-assert the interrupt state because catching the exception
cleared it
                Thread.currentThread().interrupt();
            }
        }
        System.out.println("  [InterruptWorker] ✅ Thread stopped cleanly.");
    });

    interruptWorker.start();
    Thread.sleep(1000); // Let the thread enter its 10-second sleep

    System.out.println("[Main] Sending native worker.interrupt()...");
    interruptWorker.interrupt(); // Instantly breaks the 10-second sleep block!

    System.out.println("[Main] Waiting to see if InterruptWorker dies...");
    interruptWorker.join(2000);

    if (!interruptWorker.isAlive()) {
        System.out.println("[Main] ✅ SUCCESS: InterruptWorker terminated
immediately.");
    }
  }
}
```

From: McDonagh, Sean <Sean.McDonagh@us.amentum.com>
Sent: Wednesday, June 24, 2026 2:49 PM
To: Stephen Michell <stephen.michell@maurya.on.ca>; Erhard Plödereder <ploedere@iste.uni-stuttgart.de>; tullio.vardanega@math.unipd.it; Larry Wagoner <ldw11@aol.com>
Subject: FIx

```
public class FixedThreadWorker implements Runnable {
  private final BlockingQueue<String> queue;

  public FixedThreadWorker(BlockingQueue<String> queue) {
    this.queue = queue;
  }

  @Override
  public void run() {
    // Fix: Use the native interrupt flag as your condition loop
    while (!Thread.currentThread().isInterrupted()) {
      try {
        // Fix: This blocking call will now gracefully unblock when interrupted
        String item = queue.take();
        process(item);
```

```
            } catch (InterruptedException e) {
                // Fix: Re-assert the flag because the exception cleared it!
                Thread.currentThread().interrupt();
                System.out.println("Worker thread was interrupted while blocked. Exiting
loop.");
            }
        }
        System.out.println("Clean cleanup finished here.");
    }

    private void process(String item) { /* Logic here */ }
}
```
