Java theory and practice: Dealing with InterruptedException

You caught it, now what are you going to do with it?

Brian Goetz May 23, 2006

Many Java™ language methods, such as `Thread.sleep()` and `Object.wait()`, throw `InterruptedException`. You can't ignore it because it's a checked exception, but what should you do with it? In this month's *Java theory and practice*, concurrency expert Brian Goetz explains what InterruptedException means, why it is thrown, and what you should do when you catch one.

View more content in this series

Learn more. Develop more. Connect more.

The new developerWorks Premium membership program provides an all-access pass to powerful development tools and resources, including 500 top technical titles (dozens specifically for Java developers) through Safari Books Online, deep discounts on premier developer events, video replays of recent O'Reilly conferences, and more. Sign up today.

This story is probably familiar: You're writing a test program and you need to pause for some amount of time, so you call `Thread.sleep()`. But then the compiler or IDE balks that you haven't dealt with the checked `InterruptedException`. What is `InterruptedException`, and why do you have to deal with it?

The most common response to `InterruptedException` is to swallow it -- catch it and do nothing (or perhaps log it, which isn't any better) -- as we'll see later in Listing 4. Unfortunately, this approach throws away important information about the fact that an interrupt occurred, which could compromise the application's ability to cancel activities or shut down in a timely manner.

Blocking methods

When a method throws `InterruptedException`, it is telling you several things in addition to the fact that it can throw a particular checked exception. It is telling you that it is a *blocking* method and that it will make an attempt to unblock and return early -- if you ask nicely.

A blocking method is different from an ordinary method that just takes a long time to run. The completion of an ordinary method is dependent only on how much work you've asked it to do and

© Copyright IBM Corporation 2006

whether adequate computing resources (CPU cycles and memory) are available. The completion of a blocking method, on the other hand, is also dependent on some external event, such as timer expiration, I/O completion, or the action of another thread (releasing a lock, setting a flag, or placing a task on a work queue). Ordinary methods complete as soon as their work can be done, but blocking methods are less predictable because they depend on external events. Blocking methods can compromise responsiveness because it can be hard to predict when they will complete.

Because blocking methods can potentially take forever if the event they are waiting for never occurs, it is often useful for blocking operations to be *cancelable*. (It is often useful for long-running non-blocking methods to be cancelable as well.) A cancelable operation is one that can be externally moved to completion in advance of when it would ordinarily complete on its own. The interruption mechanism provided by `Thread` and supported by `Thread.sleep()` and `Object.wait()` is a cancellation mechanism; it allows one thread to request that another thread stop what it is doing early. When a method throws `InterruptedException`, it is telling you that if the thread executing the method is interrupted, it will make an attempt to stop what it is doing and return early and indicate its early return by throwing `InterruptedException`. Well-behaved blocking library methods should be responsive to interruption and throw `InterruptedException` so they can be used within cancelable activities without compromising responsiveness.

Thread interruption

Every thread has a Boolean property associated with it that represents its *interrupted status*. The interrupted status is initially false; when a thread is interrupted by some other thread through a call to `Thread.interrupt()`, one of two things happens. If that thread is executing a low-level interruptible blocking method like `Thread.sleep()`, `Thread.join()`, or `Object.wait()`, it unblocks and throws `InterruptedException`. Otherwise, `interrupt()` merely sets the thread's interruption status. Code running in the interrupted thread can later poll the interrupted status to see if it has been requested to stop what it is doing; the interrupted status can be read with `Thread.isInterrupted()` and can be read and cleared in a single operation with the poorly named `Thread.interrupted()`.

Interruption is a cooperative mechanism. When one thread interrupts another, the interrupted thread does not necessarily stop what it is doing immediately. Instead, interruption is a way of politely asking another thread to stop what it is doing if it wants to, at its convenience. Some methods, like `Thread.sleep()`, take this request seriously, but methods are not required to pay attention to interruption. Methods that do not block but that still may take a long time to execute can respect requests for interruption by polling the interrupted status and return early if interrupted. You are free to ignore an interruption request, but doing so may compromise responsiveness.

One of the benefits of the cooperative nature of interruption is that it provides more flexibility for safely constructing cancelable activities. We rarely want an activity to stop immediately; program data structures could be left in an inconsistent state if the activity were canceled mid-update. Interruption allows a cancelable activity to clean up any work in progress, restore invariants, notify other activities of the cancellation, and then terminate.

Dealing with InterruptedException

If throwing `InterruptedException` means that a method is a blocking method, then calling a blocking method means that your method is a blocking method too, and you should have a strategy for dealing with `InterruptedException`. Often the easiest strategy is to throw `InterruptedException` yourself, as shown in the `putTask()` and `getTask()` methods in Listing 1. Doing so makes your method responsive to interruption as well and often requires nothing more than adding `InterruptedException` to your throws clause.

Listing 1. Propagating InterruptedException to callers by not catching it

```
public class TaskQueue {
    private static final int MAX_TASKS = 1000;

    private BlockingQueue<Task> queue
        = new LinkedBlockingQueue<Task>(MAX_TASKS);

    public void putTask(Task r) throws InterruptedException {
        queue.put(r);
    }

    public Task getTask() throws InterruptedException {
        return queue.take();
    }
}
```

Sometimes it is necessary to do some amount of cleanup before propagating the exception. In this case, you can catch `InterruptedException`, perform the cleanup, and then rethrow the exception. Listing 2, a mechanism for matching players in an online game service, illustrates this technique. The `matchPlayers()` method waits for two players to arrive and then starts a new game. If it is interrupted after one player has arrived but before the second player arrives, it puts that player back on the queue before rethrowing the `InterruptedException`, so that the player's request to play is not lost.

Listing 2. Performing task-specific cleanup before rethrowing InterruptedException

```
public class PlayerMatcher {
    private PlayerSource players;

    public PlayerMatcher(PlayerSource players) {
        this.players = players;
    }

    public void matchPlayers() throws InterruptedException {
        Player playerOne, playerTwo;
         try {
             while (true) {
                 playerOne = playerTwo = null;
                 // Wait for two players to arrive and start a new game
                 playerOne = players.waitForPlayer(); // could throw IE
                 playerTwo = players.waitForPlayer(); // could throw IE
                 startNewGame(playerOne, playerTwo);
             }
         }
         catch (InterruptedException e) {
             // If we got one player and were interrupted, put that player back
             if (playerOne != null)
                 players.addFirst(playerOne);
```

```
            // Then propagate the exception
            throw e;
        }
    }
}
```

Don't swallow interrupts

Sometimes throwing `InterruptedException` is not an option, such as when a task defined by `Runnable` calls an interruptible method. In this case, you can't rethrow `InterruptedException`, but you also do not want to do nothing. When a blocking method detects interruption and throws `InterruptedException`, it clears the interrupted status. If you catch `InterruptedException` but cannot rethrow it, you should preserve evidence that the interruption occurred so that code higher up on the call stack can learn of the interruption and respond to it if it wants to. This task is accomplished by calling `interrupt()` to "reinterrupt" the current thread, as shown in Listing 3. At the very least, whenever you catch `InterruptedException` and don't rethrow it, reinterrupt the current thread before returning.

Listing 3. Restoring the interrupted status after catching InterruptedException

```
public class TaskRunner implements Runnable {
    private BlockingQueue<Task> queue;

    public TaskRunner(BlockingQueue<Task> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            while (true) {
                Task task = queue.take(10, TimeUnit.SECONDS);
                task.execute();
            }
         }
         catch (InterruptedException e) {
            // Restore the interrupted status
            Thread.currentThread().interrupt();
         }
    }
}
```

The worst thing you can do with `InterruptedException` is swallow it -- catch it and neither rethrow it nor reassert the thread's interrupted status. The standard approach to dealing with an exception you didn't plan for -- catch it and log it -- also counts as swallowing the interruption because code higher up on the call stack won't be able to find out about it. (Logging `InterruptedException` is also just silly because by the time a human reads the log, it is too late to do anything about it.) Listing 4 shows the all-too-common pattern of swallowing an interrupt:

Listing 4. Swallowing an interrupt -- don't do this

```
// Don't do this
public class TaskRunner implements Runnable {
    private BlockingQueue<Task> queue;

    public TaskRunner(BlockingQueue<Task> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
             while (true) {
                 Task task = queue.take(10, TimeUnit.SECONDS);
                 task.execute();
             }
         }
         catch (InterruptedException swallowed) {
             /* DON'T DO THIS - RESTORE THE INTERRUPTED STATUS INSTEAD */
         }
    }
}
```

If you cannot rethrow `InterruptedException`, whether or not you plan to act on the interrupt request, you still want to reinterrupt the current thread because a single interruption request may have multiple "recipients." The standard thread pool (`ThreadPoolExecutor`) worker thread implementation is responsive to interruption, so interrupting a task running in a thread pool may have the effect of both canceling the task and notifying the execution thread that the thread pool is shutting down. If the task were to swallow the interrupt request, the worker thread might not learn that an interrupt was requested, which could delay the application or service shutdown.

Implementing cancelable tasks

Nothing in the language specification gives interruption any specific semantics, but in larger programs, it is difficult to maintain any semantics for interruption other than cancellation. Depending on the activity, a user could request cancellation through a GUI or through a network mechanism such as JMX or Web Services. It could also be requested by program logic. For example, a Web crawler might automatically shut itself down if it detects that the disk is full, or a parallel algorithm might start multiple threads to search different regions of the solution space and cancel them once one of them finds a solution.

Just because a task is cancelable does not mean it needs to respond to an interrupt request *immediately*. For tasks that execute code in a loop, it is common to check for interruption only once per loop iteration. Depending on how long the loop takes to execute, it could take some time before the task code notices the thread has been interrupted (either by polling the interrupted status with `Thread.isInterrupted()` or by calling a blocking method). If the task needs to be more responsive, it can poll the interrupted status more frequently. Blocking methods usually poll the interrupted status immediately on entry, throwing `InterruptedException` if it is set to improve responsiveness.

The one time it is acceptable to swallow an interrupt is when you know the thread is about to exit. This scenario only occurs when the class calling the interruptible method is part of a `Thread`, not a `Runnable` or general-purpose library code, as illustrated in Listing 5. It creates a thread that

enumerates prime numbers until it is interrupted and allows the thread to exit upon interruption. The prime-seeking loop checks for interruption in two places: once by polling the `isInterrupted()` method in the header of the while loop and once when it calls the blocking `BlockingQueue.put()` method.

Listing 5. Interrupts can be swallowed if you know the thread is about to exit

```
public class PrimeProducer extends Thread {
    private final BlockingQueue<BigInteger> queue;

    PrimeProducer(BlockingQueue<BigInteger> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            BigInteger p = BigInteger.ONE;
            while (!Thread.currentThread().isInterrupted())
                queue.put(p = p.nextProbablePrime());
        } catch (InterruptedException consumed) {
            /* Allow thread to exit */
        }
    }

    public void cancel() { interrupt(); }
}
```

Noninterruptible blocking

Not all blocking methods throw `InterruptedException`. The input and output stream classes may block waiting for an I/O to complete, but they do not throw `InterruptedException`, and they do not return early if they are interrupted. However, in the case of socket I/O, if a thread closes the socket, blocking I/O operations on that socket in other threads will complete early with a `SocketException`. The nonblocking I/O classes in `java.nio` also do not support interruptible I/O, but blocking operations can similarly be canceled by closing the channel or requesting a wakeup on the `Selector`. Similarly, attempting to acquire an intrinsic lock (enter a `synchronized` block) cannot be interrupted, but `ReentrantLock` supports an interruptible acquisition mode.

Noncancelable tasks

Some tasks simply refuse to be interrupted, making them noncancelable. However, even noncancelable tasks should attempt to preserve the interrupted status in case code higher up on the call stack wants to act on the interruption after the noncancelable task completes. Listing 6 shows a method that waits on a blocking queue until an item is available, regardless of whether it is interrupted. To be a good citizen, it restores the interrupted status in a finally block after it is finished, so as not to deprive callers of the interruption request. (It can't restore the interrupted status earlier, as it would cause an infinite loop -- `BlockingQueue.take()` could poll the interrupted status immediately on entry and throws `InterruptedException` if it finds the interrupted status set.)

Listing 6. Noncancelable task that restores interrupted status before returning

```
public Task getNextTask(BlockingQueue<Task> queue) {
    boolean interrupted = false;
    try {
        while (true) {
            try {
                return queue.take();
            } catch (InterruptedException e) {
                interrupted = true;
                // fall through and retry
            }
        }
    } finally {
        if (interrupted)
            Thread.currentThread().interrupt();
    }
}
```

Summary

You can use the cooperative interruption mechanism provided by the Java platform to construct flexible cancellation policies. Activities can decide if they are cancelable or not, how responsive they want to be to interruption, and they can defer interruption to perform task-specific cleanup if returning immediately would compromise application integrity. Even if you want to completely ignore interruption in your code, make sure to restore the interrupted status if you catch `InterruptedException` and do not rethrow it so that the code that calls it is not deprived of the knowledge that an interrupt occurred.

Related topics

- Java Concurrency in Practice
- Java development tutorials

© Copyright IBM Corporation 2006
(www.ibm.com/legal/copytrade.shtml)
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)