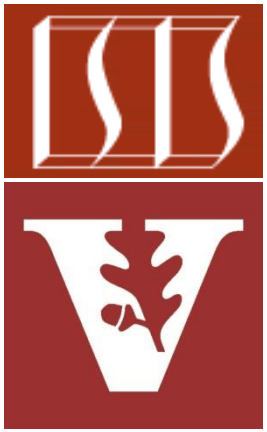
Managing the Java Thread Lifecycle: Patterns of Handling Thread Interrupts

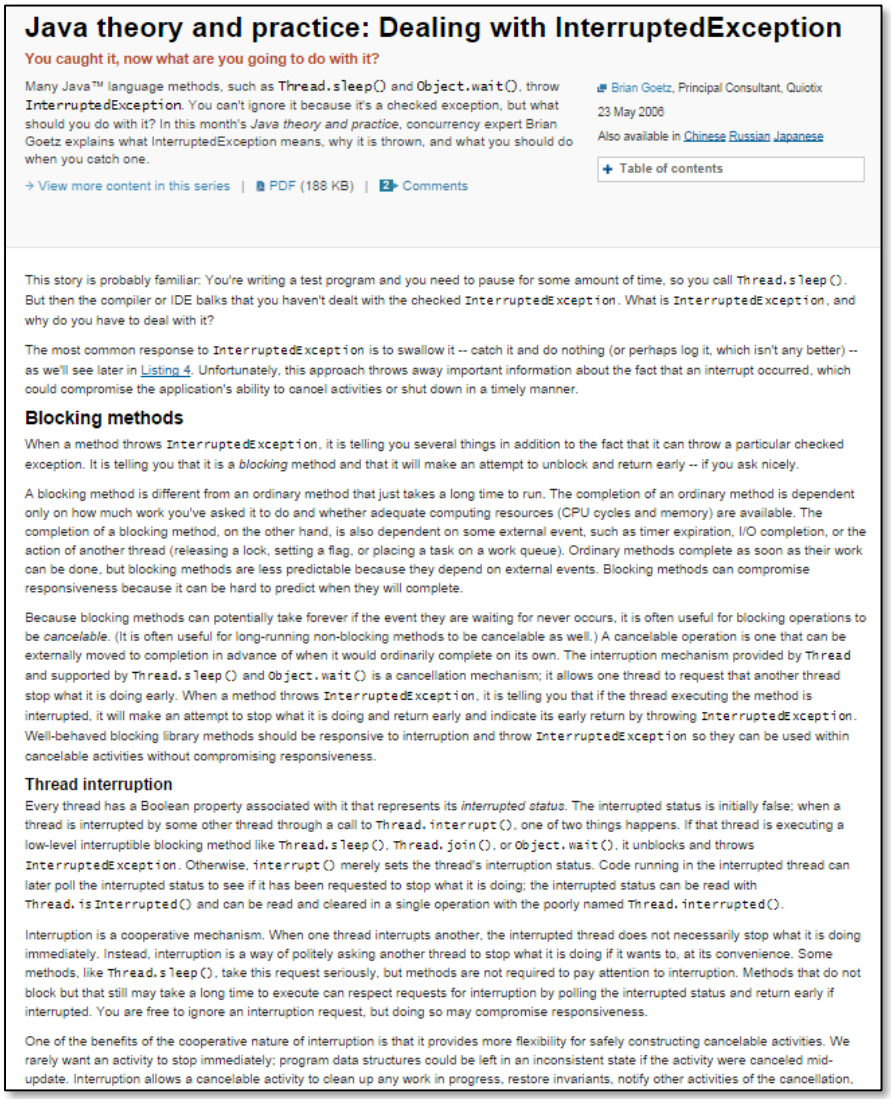


Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Know various ways to stop Java threads
  - Stopping a thread with a volatile flag
  - Stopping a thread with an interrupt request
    - Learn the patterns of handing Java thread interrupts



Java theory and practice: Dealing with InterruptedException

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

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.

Brian Goetz, Principal Consultant, Quiotix

23 May 2006

Also available in Chinese Russian Japanese

Table of contents

View more content in this series | PDF (188 KB) | Comments

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 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,

Patterns of Handling Java Thread Interrupts

Patterns of Handling Java Thread Interrupts

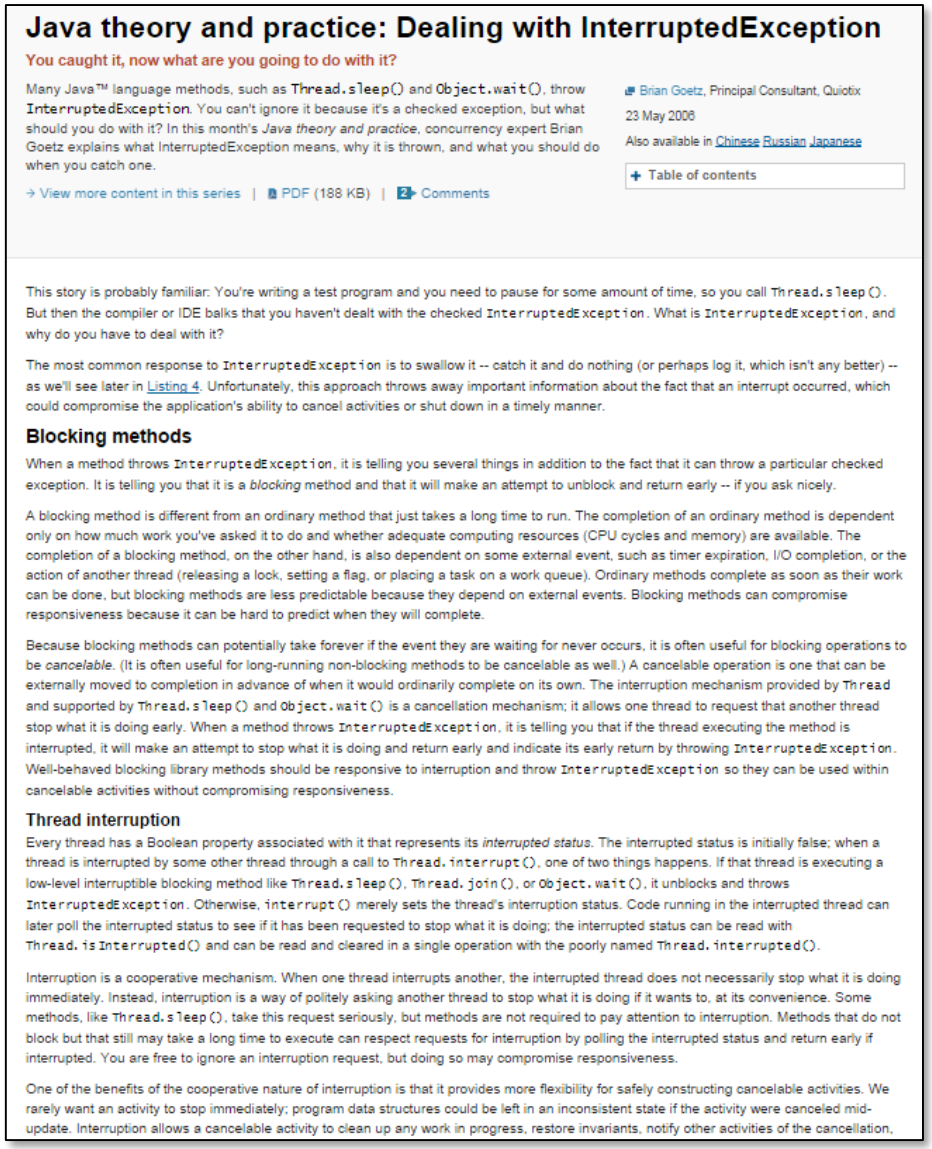
- Recall that blocking operations in Java can return automatically & throw InterruptedException if the thread is interrupted

```
void processBlocking(String args) {
  ...
  while (true) {
    try {
      Thread.currentThread().
        sleep(interval);
      synchronized(this) {
        while (someConditionFalse)
          wait();
      }
    }
    catch (InterruptedException e)
    { ... }
    ...
  }
}
```

See earlier part of the lesson on "*Stopping a Thread via an Interrupt*"

Patterns of Handling Java Thread Interrupts

- There are patterns for dealing w/Java InterruptedException



Java theory and practice: Dealing with InterruptedException

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

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.

Brian Goetz, Principal Consultant, Quiotix

23 May 2006

Also available in Chinese Russian Japanese

+ Table of contents

→ View more content in this series | PDF (188 KB) | Comments

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 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,

See www.ibm.com/developerworks/java/library/j-jtp05236/index.html?ca=drs-

Patterns of Handling Java Thread Interrupts

- There are patterns for dealing w/Java InterruptedException, e.g.
  - Propagate InterruptedException to callers by not catching it

```
public class StringBlockingQueue {
  private BlockingQueue<String>
    queue = new
    LinkedBlockingQueue<String>();

  public void put(String s)
    throws InterruptedException {
      queue.put(s);
  }

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

See docs.oracle.com/javase/8/docs/api/java/lang/InterruptedException.html

Patterns of Handling Java Thread Interrupts

- There are patterns for dealing w/Java InterruptedException, e.g.
  - Propagate InterruptedException to callers by not catching it

```
public class StringBlockingQueue {
  private BlockingQueue<String>
    queue = new
    LinkedBlockingQueue<String>();

  public void put(String s)
    throws InterruptedException {
      queue.put(s);
  }

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

The exception is explicitly listed in each method's "throw clause"

See docs.oracle.com/javase/tutorial/essential/exceptions/declaring.html

Patterns of Handling Java Thread Interrupts

- There are patterns for dealing w/Java InterruptedException, e.g.
  - Propagate InterruptedException to callers by not catching it

```
public class StringBlockingQueue {
  private BlockingQueue<String>
    queue = new
    LinkedBlockingQueue<String>();

  public void put(String s)
    throws InterruptedException {
      queue.put(s);
  }

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

```
StringBlockingQueue s =
  new StringBlockingQueue();
...
try {
  s.take();
  ...
} catch (InterruptedException e)
  ...
}
```

It's now the caller's responsibility to handle the exception properly

Patterns of Handling Java Thread Interrupts

- There are patterns for dealing w/Java InterruptedException, e.g.
  - Propagate InterruptedException to callers by not catching it
  - Perform task-specific cleanup before rethrowing

```
if (mustWait) {
  try {
    lock.wait();
  }
  catch (InterruptedException e){
    synchronized (this) {
      boolean removed =
        mWaitQueue.remove(lock);

      if (!removed)
        release();
    }
    throw e;
  }
...
```

Avoid leaking resources or leaving resources in an inconsistent state

Patterns of Handling Java Thread Interrupts

- There are patterns for dealing w/Java InterruptedException, e.g.
  - Propagate InterruptedException to callers by not catching it
  - Perform task-specific cleanup before rethrowing
  - Restore interrupted status after catching InterruptedException

```
public void doWork() {
  try {
    while (true) {
      Runnable r =
        queue.take(10, SECONDS);
      r.run();
    }
  }
  catch (InterruptedException e){
    ...
    Thread.currentThread()
      .interrupt();
  }
 }
}
```

Preserve evidence the exception occurred for use by higher levels of the call stack

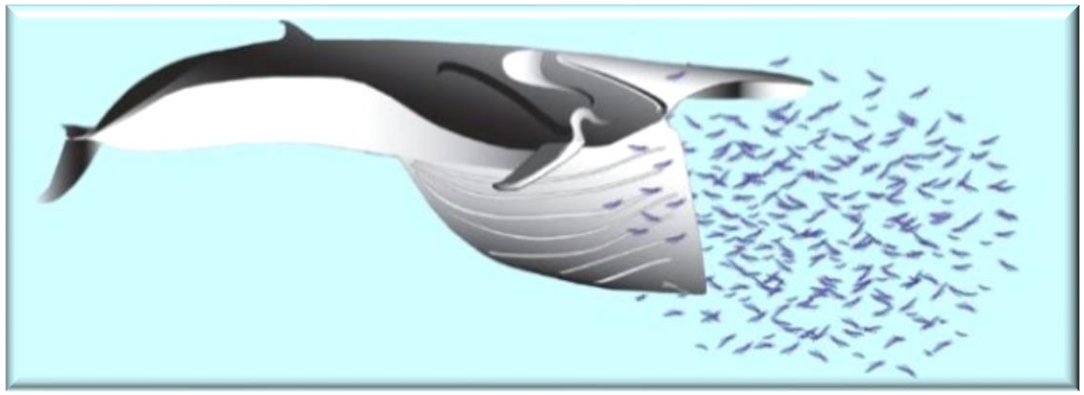
See daniel.mitterdorfer.name/articles/2015/handling-interruptedexception

Patterns of Handling Java Thread Interrupts

- There are patterns for dealing w/Java InterruptedException, e.g.
  - Propagate InterruptedException to callers by not catching it
  - Perform task-specific cleanup before rethrowing
  - Restore interrupted status after catching InterruptedException
  - Handle interrupt & "swallow" it

```
public boolean gaze() {
   try {
     int sleepTime = 1000 +
       mRandom.nextInt(4000;

     Thread.sleep(sleepTime);
     return true;
   }
   catch (InterruptedException e) {
     return false;
   }
}
```



Patterns of Handling Java Thread Interrupts

- There are patterns for dealing w/Java InterruptedException, e.g.
  - Propagate InterruptedException to callers by not catching it
  - Perform task-specific cleanup before rethrowing
  - Restore interrupted status after catching InterruptedException
  - Handle interrupt & "swallow" it

```
public boolean gaze() {
   try {
     int sleepTime = 1000 +
       mRandom.nextInt(4000;

     Thread.sleep(sleepTime);
     return true;
   }
   catch (InterruptedException e) {
     return false;
   }
}
```

e.g., often done when the thread sleep() or join() methods are called

General-purpose reusable library code should *never* swallow interrupt requests entirely (i.e., this is an "anti-pattern")

End of Managing the Java Thread Lifecycle: Patterns of Handling Thread Interrupts