DZone

InterruptedException and Interrupting Threads Explained

by Tomasz Nurkiewicz MVB · Jun. 01, 14 · Java Zone

If `InterruptedException` wasn't a checked exception, probably no one would even notice it - which would actually prevent couple of bugs throughout these years. But since it has to be handled, many handle it incorrectly or thoughtlessly. Let's take a simple example of a thread that periodically does some clean up, but in between sleeps most of the time.

```
class Cleaner implements Runnable {

  Cleaner() {
    final Thread cleanerThread = new Thread(this, "Cleaner");
    cleanerThread.start();
  }

  @Override
  public void run() {
    while(true) {
      cleanUp();
      try {
        TimeUnit.SECONDS.sleep(1);
      } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
      }
    }
  }

  private void cleanUp() {
    //...
  }

}
```

This code is wrong on so many layers!

1. Starting `Thread` in a constructor might not be a good idea in some environments, e.g. some frameworks

like Spring will create dynamic subclass to support method interception. We will end-up with two threads running from two instances.

2. `InterruptedException` is swallowed, and the exception itself is not logged properly
3. This class starts a new thread for every instance, it should use `ScheduledThreadPoolExecutor` instead, shared among many instances (more robust and memory-effective)
4. Also with `ScheduledThreadPoolExecutor` we could avoid coding sleeping/working loop by ourselves, and also switch to fixed-rate as opposed to fixed-delay behaviour presented here.
5. Last but not least there is no way to get rid of this thread, even when `Cleaner` instance is no longer referenced by anything else

All problems are valid, but swallowing `InterruptedException` is its biggest sin. Before we understand why, let us think for a while what does this exception mean and how we can take advantage of it to interrupt threads gracefully. Many blocking operations in JDK declare throwing `InterruptedException`, including:

- `Object.wait()`
- `Thread.sleep()`
- `Process.waitFor()`
- `AsynchronousChannelGroup.awaitTermination()`
- Various blocking methods in `java.util.concurrent.*`, e.g. `ExecutorService.awaitTermination()`, `Future.get()`, `BlockingQueue.take()`, `Semaphore.acquire()` `Condition.await()` and many, many others
- `SwingUtilities.invokeAndWait()`

Notice that blocking I/O does not throw `InterruptedException` (which is a shame). If all these classes declare `InterruptedException`, you might be wondering when is this exception ever thrown?

- When a thread is blocked on some method declaring `InterruptedException` and you call `Thread.interrupt()` on such thread, most likely blocked method will immediately throw `InterruptedException`.
- If you submitted a task to a thread pool (`ExecutorService.submit()`) and you call `Future.cancel(true)` while the task was being executed. In that case the thread pool will try to interrupt thread running such task for you, effectively interrupting your task.

Knowing what `InterruptedException` actually means, we are well equipped to handle it properly. If someone tries to interrupt our thread and we discovered it by catching `InterruptedException`, the most reasonable thing to do is letting said thread to finish, e.g.:

```
1  class Cleaner implements Runnable, AutoCloseable {
2
3    private final Thread cleanerThread;
4
5    Cleaner() {
6      cleanerThread = new Thread(this, "Cleaner");
       cleanerThread.start();
```

```
    cleanerThread.start();
  }

  @Override
  public void run() {
    try {
      while (true) {
        cleanUp();
        TimeUnit.SECONDS.sleep(1);
      }
    } catch (InterruptedException ignored) {
      log.debug("Interrupted, closing");
    }
  }

  //...

  @Override
  public void close() {
    cleanerThread.interrupt();
  }
}
```

Notice that `try-catch` block now surrounds `while` loop. This way if `sleep()` throws `InterruptedException`, we will break out of the loop. You might argue that we should log `InterruptedException`'s stack-trace. This depends on the situation, as in this case interrupting a thread is something we really expect, not a failure. But it's up to you. The bottom-line is that if `sleep()` is interrupted by another thread, we quickly escape from `run()` altogether. If you are very careful you might ask what happens if we interrupt thread while it's in `cleanUp()` method rather than sleeping? Often you'll come across manual flag like this:

```
private volatile boolean stop = false;

@Override
public void run() {
  while (!stop) {
    cleanUp();
    TimeUnit.SECONDS.sleep(1);
  }
}

@Override
public void close() {
  stop = true;
}
```

However notice that `stop` flag (it has to be `volatile`!) won't interrupt blocking operations, we have to wait until `sleep()` finishes. On the other side one might argue that explicit `flag` gives us better control since we can monitor its value at any time. It turns out thread interruption works the same way. If someone interrupted

can monitor its value at any time. It turns out thread interruption works the same way. If someone interrupted thread while it was doing non-blocking computation (e.g. inside `cleanUp()`) such computations aren't interrupted immediately. However thread is marked as *interrupted* and every subsequent blocking operation (e.g. `sleep()`) will simply throw `InterruptedException` immediately - so we won't loose that signal.

We can also take advantage of that fact if we write non-blocking thread that still wants to take advantage of thread interruption facility. Instead of relying on `InterruptedException` we simply have to check for `Thread.isInterrupted()` periodically:

```
1  public void run() {
2    while (Thread.currentThread().isInterrupted()) {
3      someHeavyComputations();
4    }
5  }
```

Above, if someone interrupts our thread, we will abandon computation as soon as `someHeavyComputations()` returns. If it runs for two long or infinitely, we will never discover interruption flag. Interestingly `interrupted` flag is not a *one-time pad*. We can call `Thread.interrupted()` instead of `isInterrupted()`, which will reset `interrupted` flag and we can continue. Occasionally you might want to ignore interrupted flag and continue running. In that case `interrupted()` might come in handy. BTW I (imprecisely) call "getters" that change the state of object being observed " *Heisengetters*".

Note on `Thread.stop()`

If you are old-school programmer, you may recall `Thread.stop()` method, which has been deprecated for 10 years now. In Java 8 there were plans to "de-implement it", but in 1.8u5 it's still there. Nevertheless, don't use it and refactor any code using `Thread.stop()` into `Thread.interrupt()`.

`Uninterruptibles` from Guava

Rarely you might want to ignore `InterruptedException` altogether. In that case have a look at `Uninterruptibles` from Guava. It has plenty of utility methods like `sleepUninterruptibly()` or `awaitUninterruptibly(CountDownLatch)`. Just be careful with them. I know they don't declare `InterruptedException` (which might be handful), but they also completely prevent current thread from being interrupted - which is quite unusual.

Summary

By now I hope you have some understanding why certain methods throw `InterruptedException`. The main takeaways are:

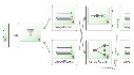
- Caught `InterruptedException` should be handled *properly* - most of the time it means breaking out of the current task/loop/thread entirely
- Swallowing `InterruptedException` is rarely a good idea

- If thread was interrupted while it wasn't in a blocking call, use `isInterrupted()`. Also entering blocking method when thread was already interrupted should immediately throw `InterruptedException`

Like This Article? Read More From DZone

Drools Decision Tables with Camel and Spring

Easy Messaging with STOMP over WebSockets using ActiveMQ and HornetQ



Message Processing With Spring Integration

Free DZone Refcard
Core Java Concurrency

Topics: JAVA , ENTERPRISE-INTEGRATION , TIPS AND TRICKS

Java Partner Resources

Java Quiz 9: Demonstrating Multilevel Inheritance

by Sar Maroof · Jan 18, 18 · Java Zone

Before we start with this week's quiz, here is the answer to Java Quiz 8: Upcasting and Downcasting Objects.

By upcasting objects, the overridden variable depends on the type of the object reference vc, but the overridden methods depend on the type of the object that was created. By downcasting objects, both variables and methods depend on the type of the object reference car.

The correct answer is: e.

Here is the quiz for today!

What happens when the following program is compiled and run?

Note: The classes Animal, WildAnimal, and BigCat are three separate files in one package.

Animal.java:

```
public class Animal {
    Animal() {
        System.out.print("Tiger" + " ");
    }
}
```

WildAnimal.java:

```
class WildAnimal extends Animal {
    WildAnimal(String s) {
        System.out.print(s + " ");
    }
}
```

BigCat.java:

```
public class BigCat extends WildAnimal {
    BigCat() {
        this("Jaguar");
    }
    BigCat(String s) {
        super(s);
        System.out.print(s + " ");
    }
    public static void main(String[] args) {
        new WildAnimal("Leopard");
        new BigCat();
    }
}
```

- A. This program writes "Tiger Leopard Jaguar Jaguar" to the standard output.
- B. This program writes "Tiger Leopard Tiger Jaguar Jaguar" to the standard output.
- C. This program writes "Tiger Leopard Jaguar" to the standard output.
- D. This program writes "Tiger Leopard Tiger Jaguar" to the standard output.
- E. This program writes "Tiger Leopard" to the standard output.
- F. This program does not compile.

The correct answer and its explanation will be included in the next quiz in two weeks! For more Java quizzes, puzzles, and assignments, take a look at my site!

Like This Article? Read More From DZone

Java Quiz: Extending an Abstract Class

Java Quiz: Using TreeMap in Java

Java Quiz: Nested Classes

Free DZone Refcard
Core Java Concurrency

Topics: JAVA, QUIZ