How to Handle an InterruptedException

by Yegor Bugayenko MVB · Oct. 20, 15 · Java Zone · Tutorial

`InterruptedException` is a permanent source of pain in Java, for junior developers especially. But it shouldn't be. It's a rather simple and easy-to-understand idea. Let me try to describe and simplify it.

Let's start with this code:

```
while (true) {
  // Nothing
}
```

What does it do? Nothing, it just spins the CPU endlessly. Can we terminate it? Not in Java. It will only stop when the entire JVM stops, when you hit `Ctrl-C`. There is no way in Java to terminate a thread unless the thread exits by itself. That's the principle we have to have in mind, and everything else will just be obvious.

Let's put this endless loop into a thread:

```
Thread loop = new Thread(
  new Runnable() {
    @Override
    public void run() {
      while (true) {
      }
    }
  }
);
loop.start();
// Now how do we stop it?
```

So, how do we stop a thread when we need it to stop?

Here is how it is designed in Java. There is a flag in every thread that we can set from the outside. And the thread may check it occasionally and stop its execution. Voluntarily! Here is how:

```
Thread loop = new Thread(
  new Runnable() {
    @Override
    public void run() {
      while (true) {
        if (Thread.interrupted()) {
          break;
        }
        // Continue to do nothing
      }
    }
  }
);
loop.start();
loop.interrupt();
```

This is the only way to ask a thread to stop. There are two methods that are used in this example. When I call `loop.interrupt()` , a flag is set to `true` somewhere inside the thread `loop` . When I call `interrupted()` , the flag is returned and immediately set to `false` . Yeah, that's the design of the method. It checks the flag, returns it, and sets it to `false` . It's ugly, I know.

Thus, if I never call `Thread.interrupted()` inside the thread and don't exit when the flag is `true` , nobody will be able to stop me. Literally, I will just ignore their calls to `interrupt()` . They will ask me to stop, but I will ignore them. They won't be able to interrupt me.

Thus, to summarize what we've learned so far, a properly designed thread will check that flag once in a while and stop gracefully. If the code doesn't check the flag and never calls `Thread.interrupted()` , it accepts the fact that sooner or later it will be terminated cold turkey, by clicking `Ctrl-C` .

Sound logical so far? I hope so.

Now, there are some methods in JDK that check the flag for us and throw `InterruptedException` if it is set. For example, this is how the method `Thread.sleep()` is designed (taking a very primitive approach):

```
public static void sleep(long millis)
  throws InterruptedException {
  while (/* You still need to wait */) {
    if (Thread.interrupted()) {
      throw new InterruptedException();
    }
    // Keep waiting
  }
}
```

Why is it done this way? Why can't it just wait and never check the flag? Well, I believe it's done for a good reason. And the reason is the following (correct me if I'm wrong): The code should either be bullet-fast or interruption-ready, nothing in between.

If your code is fast, you never check the interruption flag, because you don't want to deal with any interruptions. If your code is slow and may take seconds to execute, make it explicit and handle interruptions somehow.

That's why `InterruptedException` is a checked exception. Its design tells you that if you want to pause for a few milliseconds, make your code interruption-ready. This is how it looks in practice:

```
try {
  Thread.sleep(100);
} catch (InterruptedException ex) {
  // Stop immediately and go home
}
```

Well, you could let it float up to a higher level, where they will be responsible for catching it. The point is that someone will have to catch it and do something with the thread. Ideally, just stop it, since that's what the flag is about. If `InterruptedException` is thrown, it means someone checked the flag and our thread has to finish what it's doing ASAP.

The owner of the thread doesn't want to wait any longer. And we must respect the decision of our owner.

Thus, when you catch `InterruptedException` , you have to do whatever it takes to wrap up what you're doing and exit.

Now, look again at the code of `Thread.sleep()` :

```
public static void sleep(long millis)
  throws InterruptedException {
  while (/* ... */) {
    if (Thread.interrupted()) {
      throw new InterruptedException();
    }
  }
}
```

Remember, `Thread.interrupted()` not only returns the flag but also sets it to `false` . Thus, once `InterruptedException` is thrown, the flag is reset. The thread no longer knows anything about the interruption request sent by the owner.

The owner of the thread asked us to stop, `Thread.sleep()` detected that request, removed it, and threw `InterruptedException` . If you call `Thread.sleep()` , again, it will not know anything about that interruption request and will not throw anything.

See what I'm getting at? It's very important not to lose that `InterruptedException` . We can't just swallow it and move on. That would be a severe violation of the entire Java multi-threading idea. Our owner (the owner of our thread) is asking us to stop, and we just ignore it. That's a very bad idea.

This is what most of us are doing with `InterruptedException` :

```
try {
  Thread.sleep(100);
} catch (InterruptedException ex) {
  throw new RuntimeException(ex);
}
```

It looks logical, but it doesn't guarantee that the higher level will actually stop everything and exit. They may just catch a runtime exception there, and the thread will remain alive. The owner of the thread will be disappointed.

We have to inform the higher level that we just caught an interruption request. We can't just throw a runtime exception. Such behavior would be too irresponsible. The entire thread received an interruption request, and we merely swallow it and convert it into a `RuntimeException` . We can't treat such a serious situation so loosely.

This is what we have to do:

```
try {
  Thread.sleep(100);
} catch (InterruptedException ex) {
  Thread.currentThread().interrupt(); // Here!
  throw new RuntimeException(ex);
}
```

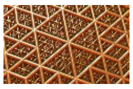
We're setting the flag back to `true` !

Now, nobody will blame us for having an irresponsible attitude toward a valuable flag. We found it in `true` status, cleared it, set it back to `true` , and threw a runtime exception. What happens next, we don't care.

I think that's it. You can find a more detailed and official description of this problem here: Java Theory and Practice: Dealing With InterruptedException.

Like This Article? Read More From DZone

Concurrency With Akka


Five Ways to Update Fields in a Thread Safe Way: Part 1

How I Test My Java Classes for Thread Safety

Free DZone Refcard
Getting Started With Vaadin 10

Topics: JAVA , CONCURRENCY , THREADING , INTERRUPT

Published at DZone with permission of Yegor Bugayenko , DZone MVB. See the original article here.
Opinions expressed by DZone contributors are their own.

ava Partner Resources

va Ecosystem 2018 Infographic by JetBrains
tBrains

ıilding Reactive Microservices in Java: Asynchronous and Event-Based Application Design
ɜd Hat Developer Program

|

grating to Microservice Databases
ed Hat Developer Program

|

eveloping Reactive Microservices: Enterprise Implementation in Java
ghtbend

|