When Is A Java Exception Not An Exception?

by Julian Robichaux, panagenda
originally published on socialbizug.org, May 2012

When is a Java Exception not an Exception? When it's an Error, of course. Need more explanation? Read on.

The Problem

I had some Java code I wanted to test as an XAgent (for more on XAgents, see: http://www.mindoo.com/web/blog.nsf/dx/17.07.2011101855KLEBRW.htm?opendocument&comments). So I wrapped up the code in a small method like this:

```
public static int wrapperMethod () {
  try {
    return doSomething();
  } catch (Exception e) {
    return -1;
  }
}
```

and I wrote some Server-side JavaScript (SSJS) on an XPage like this:

```
try {
  var result = com.example.MyObject.wrapperMethod();
  print("result = " + result);
} catch (e) {
  _dump(e);
}
```

There was more to it, but that'll get us where we need to go.

The problem I ran into is that the Java code was not running properly, but I wasn't able to catch the error anywhere. It wasn't being reported in the Java try/catch block or the XPages try/catch block. I tried printing stack traces, I tried logging, I tried yelling angry things at the server console... nothing seemed to work. The code was bombing out and I couldn't get anything to tell me where or why.

When I attached a debugger and stepped through the code, I found myself in the middle of an `InvocationTargetException` at the line of code that was failing. I thought this was odd at first, but then I realized that it was because of the way XPages uses

reflection in order to run Java code -- an `InvocationTargetException` happens when a reflected Java method has an issue. (As an interesting sidenote, see this short discussion on how XPages uses classloaders: http://stackoverflow.com/questions/5352550/meaning-of-java-lang-classcastexception-someclass-incompatible-with-someclass)

I wasn't getting back an `InvocationTargetException` though. I was getting the error that was wrapped up inside that Exception, which was an `ExceptionInInitializerError`. This is the kind of thing you get back if there is a problem initializing a static variable inside a class. The interesting thing about an `ExceptionInInitializerError` is: it's not a Java Exception. So it won't get caught by a standard try/catch block.

Throwables, Exceptions, and Errors

When Java code has a problem, a Throwable object is supposed to be generated. The two subclasses of Throwable are Exception and Error, and there are many subclasses beneath them. There is some very detailed consideration of general Java error handling theory in both the Java Virtual Machine Specification (http://docs.oracle.com/javase/specs/jvms/se5.0/html/Concepts.doc.html#22727) and the Java Language Specification (http://docs.oracle.com/javase/specs/jls/se5.0/html/exceptions.html). To avoid completely oversimplifying the topic by boiling down several pages worth of commentary into a few sentences, I will leave those links for you as reference and historical background, and instead quote from the Exception and Error JavaDocs:

> "The class `Exception` and its subclasses are a form of `Throwable` that indicates conditions that a reasonable application might want to catch."
>
> "An `Error` is a subclass of `Throwable` that indicates serious problems that a reasonable application should **not** try to catch. Most such errors are abnormal conditions."

In other words, Exceptions are supposed to be handled by your application; Errors are supposed to be ignored so that the JVM can deal with them. The exception to this rule (pun intended) is the `RuntimeException` and its subclasses, which are things like `NullPointerException` that *can* be caught but don't *have* to be.

Java Error objects are typically supposed to be ignored because, to quote the Java Language Specification, they are conditions for which "recovery is typically not possible". Subclasses of Error include `OutOfMemoryError` and `ClassCircularityError`. You certainly wouldn't want to catch an `OutOfMemoryError` and then decide "well, I'll just ignore that for now and keep going." It's best to let those sorts of things filter up to the JVM.

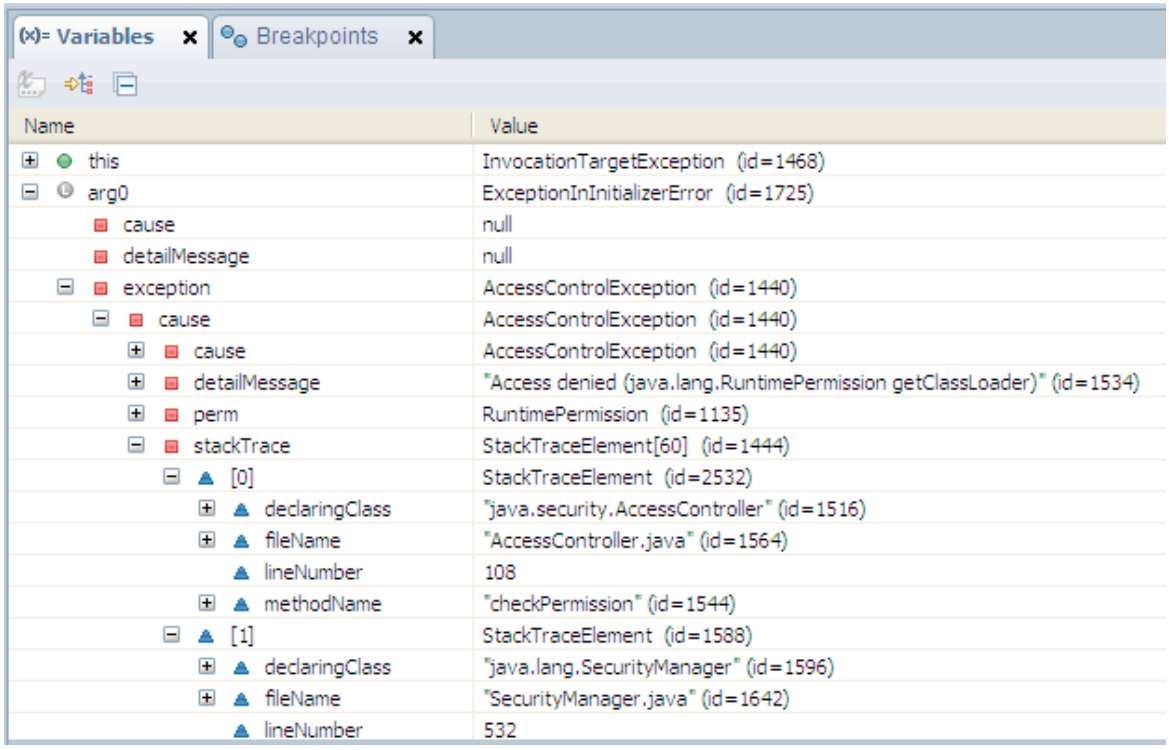
The Java language designers made a distinction between Exceptions and Errors so you can write code like:

```
try {
  doSomething();
} catch (Exception e) {
  runAndHide();
}
```

and not have to filter out the “normal” exceptions from the “devastating” errors. The typical try/catch block will only catch Exceptions, which is all you're supposed to worry about (in code) in the first place. 99% of the time that's exactly how you want it to work, and if you start catching Errors too you're just coding yourself into trouble.

My Problem, and its Solution

Back to my problem. I found myself in that 1% of the time where I actually wanted to catch an Error, because in this case I was suffering from a poorly-written static method in an open-source library I was calling. The debugger output looked like this:



You can see that there is:

- an `InvocationTargetException` from XPages calling the Java code, which contains:
- an `ExceptionInInitializerError` from a static method with an uncaught exception, which **then** contains:
- an `AccessControlException`

That `AccessControlException` -- which is one of those `RuntimeExceptions` that you are not required to check in your code -- was what I really wanted to know about. That ended up being the root of my problem, as the method causing the problem was trying to do something that the Java `SecurityManager` wasn't allowing it to do.

It almost always ends up being a security issue, doesn't it?

When I changed my code to this:

```
public static int wrapperMethod () {
  try {
    return doSomething();
  } catch (Exception e) {
    return -1;
  } catch (Throwable t) {
    return -2;
  }
}
```

suddenly I was able to catch the error and do something with it (print the stack trace, re-throw it, work around it, or whatever). Again, that's just a rough example, not what I actually ended up doing.

There Be Dragons...
That's not really the end of the story though. You might come away from this thinking that you should recode your Java apps to catch all Throwables instead of only Exceptions, just in case. That would be incorrect. I need to warn you that writing your Java code to catch all Throwables (Exceptions and Errors) is almost always a bad idea.

The bigger lessons to learn here are:

- When in doubt, use a debugger
- Code your static methods very defensively

- “catch” doesn't always catch everything, in Java or XPages

On that note, if you want to learn more about debugging Java in XPages, please see this video by Niklas Heidloff (http://www.openntf.org/blogs/openntf.nsf/d6plinks/NHEF-8CCDTQ) or the Lotusphere Show-n-Tell presentation that Mark Myers and I did a few months ago (http://londondevelopercoop.com/ldc.nsf/pages/goodies).