Adventures in Groovy – Part 35: Error Trapping Groovy Calculations

There has not been alot of troubleshooting discussed in the adventures series. Just like with most coding languages, you can gracefully handle errors resulting from actions (like divide by 0) and return descriptive information to the users and administrators in the job console. There are several benefits that I see.

- As previously stated, since the error is accounted for, the user doesn’t get a message that shows a failure with no context.
- The error object will provide more information about what happened and what should be done to fix it in the future.
- Predefined actions can take place since the error doesn’t interrupt the script, like returning an error message that tells the user to contact the administrator with an action

Error Handling Introduction

Try / catch / finally is a concept most development languages have. Conceptually, you “try” some group of commands and “catch” any errors that might happen. If you “catch” an error, you account for it by doing something. “Finally,” you perform any closing actions.

```
try {
  def arr = 1/0
} catch(Exception ex) {
  println ex.toString()
  println ex.getMessage()
  println ex.getStackTrace()
}finally {
   println "The final block"
}
```

In this case, ex.toString() prints

```
java.lang.ArithmeticException: Division by zero
```

ex.getMessage() prints

```
Division by zero
```

and ex.getStackTrace()

```
[java.math.BigDecimal.divide(Unknown Source),
org.codehaus.groovy.runtime.typehandling.BigDecimalMath.divide
Impl(BigDecimalMath.java:68),
org.codehaus.groovy.runtime.typehandling.IntegerMath.divideImp
l(IntegerMath.java:49),
org.codehaus.groovy.runtime.dgmimpl.NumberNumberDiv$NumberNumb
er.invoke(NumberNumberDiv.java:323),
org.codehaus.groovy.runtime.callsite.PojoMetaMethodSite.call(P
ojoMetaMethodSite.java:56),
org.codehaus.groovy.runtime.callsite.CallSiteArray.defaultCall
(CallSiteArray.java:48),
org.codehaus.groovy.runtime.callsite.AbstractCallSite.call(Abs
tractCallSite.java:113),
org.codehaus.groovy.runtime.callsite.AbstractCallSite.call(Abs
tractCallSite.java:125),
ConsoleScript11.run(ConsoleScript11:2),
groovy.lang.GroovyShell.runScriptOrMainOrTestOrRunnable(Groovy
Shell.java:263),
groovy.lang.GroovyShell.run(GroovyShell.java:387),
groovy.lang.GroovyShell.run(GroovyShell.java:366),
groovy.lang.GroovyShell.run(GroovyShell.java:170),
groovy.lang.GroovyShell$run$0.call(Unknown Source),
groovy.ui.Console$_runScriptImpl_closure18.doCall(Console.groo
vy:1123),
groovy.ui.Console$_runScriptImpl_closure18.doCall(Console.groo
vy), sun.reflect.NativeMethodAccessorImpl.invoke0(Native
Method), sun.reflect.NativeMethodAccessorImpl.invoke(Unknown
Source),
sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown
Source), java.lang.reflect.Method.invoke(Unknown Source),
org.codehaus.groovy.reflection.CachedMethod.invoke(CachedMetho
d.java:98),
groovy.lang.MetaMethod.doMethodInvoke(MetaMethod.java:325),
org.codehaus.groovy.runtime.metaclass.ClosureMetaClass.invokeM
ethod(ClosureMetaClass.java:294),
groovy.lang.MetaClassImpl.invokeMethod(MetaClassImpl.java:989)
,     groovy.lang.Closure.call(Closure.java:415),
groovy.lang.Closure.call(Closure.java:409),
groovy.lang.Closure.run(Closure.java:496),
java.lang.Thread.run(Unknown Source)]
```

The script in the final block is also written out. It is intended for cleanup and tasks that run at the end of a script regardless of whether there is an error or not.

Handling Errors Uniquely

The catch command can be replicated to handle errors uniquely. Let's expand on the example above. Assume the variable is coming from an RTP or cell value. The following has a catch for a specific error. The java.lang.ArithmeticException is equal to the output of ex.toString(). There are probably thousands of errors, if not more. The easiest way for me to grab these is to use the ex.toString() and see what it produces. I have no desire to remember or learn them all!

The following will do something different for the divide by zero error than all other errors.

```
try
  {
  def denominator = 0
  println 1/denominator
  }
catch(java.lang.ArithmeticException ex)
  {
  println ex.getMessage()
  println "an action should be taken to account for the error"
  }
catch(Exception ex)
  {
  println ex.toString()
  println ex.getMessage()
  println ex.getStackTrace()
  }
finally
  {
  println "The final block"
  }
```

Finishing Up

This requires a little more effort, but once you get used to doing this, it can be reused. I hear the argument that if you account for every possible situation, you don’t need to error trap. That is true, and if you are smart enough to account for everything that can go wrong, don’t include this in your Groovy calculation. I would argue that simple calculations probably don’t need this level of error handling, but more complex logic might be a candidate. The example above could obviously be handled with an if statement, but put it in context. It is used to illustrate the concept of try / catch / finally.