Adventures in Groovy – Part 52: And You Thought Essbase Only Stored Numbers

My 20+ years of using Essbase I was told, and had no reason not to believe, only stored numbers. We obviously have lists and with Planning it appears we have text. If you aren't familiar with how this works, the strings are stored in the Planning repository and the index is stored in Essbase. If you thought the same thing, you were as wrong as I was.

What is NaN

I have been learning and implementing Groovy solutions for 2-3 years now and came across something I now only have never seen, but didn't think was possible. Java, and therefore Groovy, has a concept of NaN. NaN stands for Not A Number. NaN is the result of mathematical operators that create non numbers. Log, square root, division, and I am sure plenty of other formulas that I learned before I was 15 and long forgot, can result in what Java interprets as non numeric values. The two that I have found are NaN and Infinity. An example of 4/0 would result in NaN. 0/4 would result in Infinity.

NaN in Groovy

Prior to about 2 months ago, I accounted for these scenarios in my logic and never had an issue. Recently, in writing some basic math, like revenue / units, I didn't account for the possibility that revenue or units would be a zero. If these scenarios are tested in Groovy, errors are encountered and honestly, I thought my logic in a business rule would have

produced a divide by 0 error when the denominator was a 0.

```
java.lang.ArithmeticException: Division by zero
at ConsoleScript2.run(ConsoleScript2:1)
at
java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke
0(Native Method)
at
java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke
(NativeMethodAccessorImpl.java:62)
at
java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.in
voke(DelegatingMethodAccessorImpl.java:43)
```

I thought, like in Essbase, 4/0 would result in a 0. I found out the hard way that is not the case!

Types Of NaNs and Infinities

In my case, I didn’t care of the sub type of Nan or Infinity the results was, just that it happened and I needed to account for it. These can be checked very simply.

```
double simpleSample = 4/0
if( simpleSample.inNaN() || simpleSample.isInfinite() ) {
  println 'ERROR'
}
```

Your situation might be different. If it is, these are the types I am aware of that you can check for

```
//NaN variances
isNaN(nan)
isNaN(zero_div_zero)
isNaN(sqrt_negative);
isNaN(inf_minus_inf);
isNaN(inf_times_zero);
isNaN(quiet_nan1);
isNaN(quiet_nan2);
isNaN(signaling_nan1);
isNaN(signaling_nan2);
isNaN(nan_minus);
```

```
isNaN(log_negative);
isNaN(positive_inf);
isNaN(negative_inf);
 // Infinite variances
isInfinite(positive_inf);
isInfinite(negative_inf);
```

What Do You See In Planning and Essbase

So here is where I really was confused! Everything I I thought I knew was wrong.

Surprise Number One

If either of these conditions occurs, the cell that was calculated in Groovy and stored in Planning/Essbase is actually stored differently. I can't say for sure what happens on the back end, but when the data is exported, rather than a numeric value, it will export NaN. Yes, you will see something like 10,20,20,NaN,40….

Surprise Number Two

If either of these conditions occurs, the cell that was calculated in Groovy and stored in Planning/Essbase shows a number that makes no sense in a data form when opened in Smart View. A value of 65535 will be displayed. This value can be edited/changed. If it is the source of another member formula or calculation, it will also show a value of 65535.

Surprise Number Three

The same thing is NOT what you see in a data form opened in the UI. In the UI (web version), NaN or Infinity will actually be displayed in the effected cell. This almost makes sense if I didn't see 65535 in Smart View.

Stop NaNs From Happening

There are probably a million ways to handle this. For what it

is worth, I want to share how I handled it and why. First, I created a function in my calculation that accepted one parameter, which was the value in which I was evaluating for Nan or Infinity. Inside this I used an Elvis operator and returned 0 if it was Nan or Infinity, and the value submitted to the function if it was a numeric value. The reason I created a function was because I had more than 30 formulas that I needed to check for this and it was easier to write the code once.

```
double nanCheck(double input){ (input.isNaN() || input.isInfinite()) ? 0 : input }

// Use Case Example
DataCell rate
DataCell units
operation.grid.dataCellIterator({DataCell cell -> cell.edited}).each{cell->
    rate = cell
    if(rate.accountName == 'Small_Unit_Cost'){
        units = cell.crossDimCell('Small_Units')
                                            units.data = nanCheck(cell.crossDimCell('Revenue').data / rate.data)
    }
    else if(...)
        {...}
}
```

That’s A Wrap

it is really important to account for this for obvious reasons. If you are testing for NaN and Infinity, save yourself some trouble and if there is a possibility of it occurring, start doing it now. It is a pain to strip it out afterwards if it gets into a UAT situation or even Production. One last thing. If you are looking at this and thinking, this should really return #Missing. You surely can do that. There are a few changes that have to be made. First, the function can’t be double. Since #Missing is a string, it would need to

be a string. The second issue is that you can’t set data, which is a double, to a string. You would have to use formattedValue. The changes would look something like this.

```
String nanCheck(double input){ (input.isNaN() || input.isInfinite()) ? '#Missing' : input }

// Use Case Example
DataCell rate
DataCell units
operation.grid.dataCellIterator({DataCell cell -> cell.edited}).each{cell->
    rate = cell
    if(rate.accountName == 'Small_Unit_Cost'){
        units = cell.crossDimCell('Small_Units')
                                units.formattedValue = nanCheck(cell.crossDimCell('Revenue').data / rate.data)
    }
    else if(...)
        {...}
}
```