Chapter Five:
Fault Injection

Changes Implemented From Last Chapter

In the last chapter we had attempted to resolve the issue of the script having information on the test cases. We had approached the problem from the perspective of the methods, instead of the test cases, so we were preloading the method names in and building the script around the methods. We again missed the mark here, but luckily, the third time's the charm and we were able to take the fundamentals of how our script ran and use that to implement the testing framework correctly. We just needed to practice our reading comprehension; it's a learning experience.

Updated Script: Test Case Loading/Retrieval

```
testCaseNames = []

# Call the ls command on the testCases directory
os.system("ls testCases > temp.txt")

# Open the temp file
tempFile = open("temp.txt", "r")
```

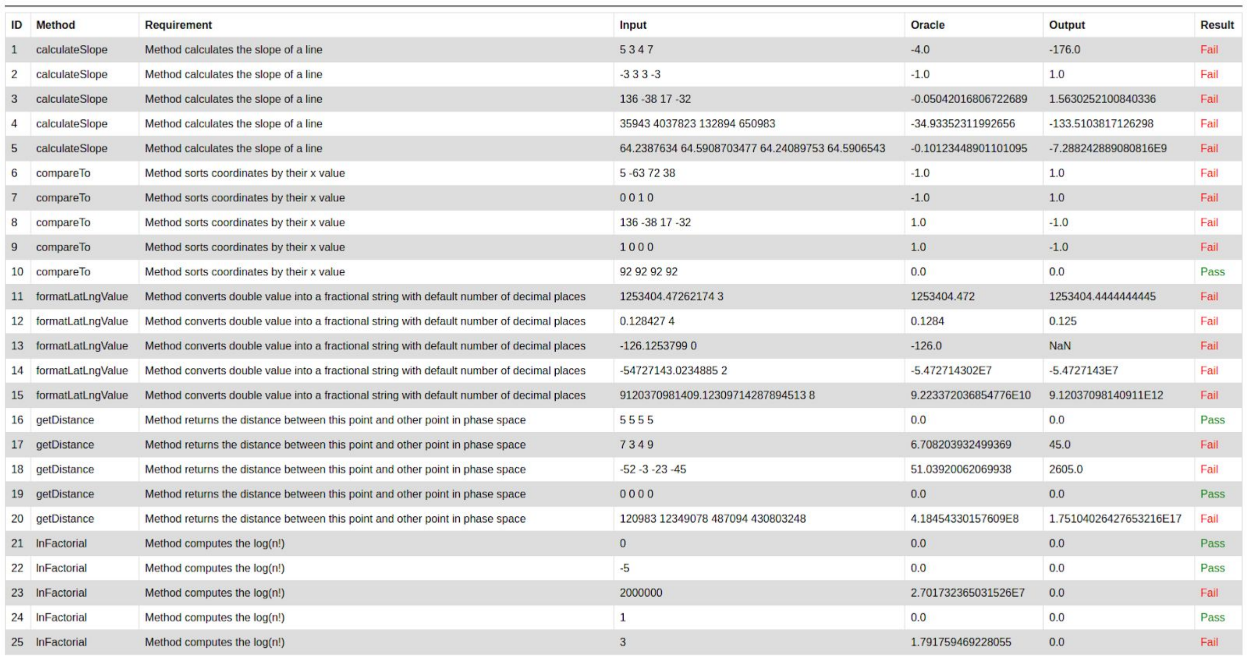
Unlike the last version of our script, the new and improved, and most importantly, correct, version, gets the test cases from their directory. The only potential problem with this approach of preloading the test cases, is the potentiality of having a *lot* of test cases. This would exponentially slow the script down to preload all of those test cases and then execute each one individually. But for the 25 test cases, it is acceptable.

Updated Script: Test Case Execution

```
# Run each test case
for testCase in testCaseNames:
    jsonFile = readJsonAtLocation("testCases/" + testCase)
    print("Running test " + str(jsonFile["id"]))
    moveProjectFileandCompile(jsonFile)
    runTestCase(jsonFile)
    cleanUpTestCaseExe(jsonFile)
    writeTestResults("temp/" + jsonFile["method"] + "TestCase" + str(jsonFile["id"]) + "results.txt", jsonFile)
    os.system("rm ./temp/" + jsonFile["method"] + "TestCase" + str(jsonFile["id"]) + "results.txt")
```

Because our script was executing in terms of the method names, the output was being grouped by method name, because that's how the test cases were organized. The problem with this approach is, if the order of the test cases changed the output wouldn't be grouped correctly. Thus, when we fixed the script and it met the specifications, we also had to reimplement how the output would be organized. So we decided to order it in ascending order by test ID. For now, the test cases for each method are grouped together, just because that's how the test cases exist in their directory, but if test cases were added it would remain sorted by test case ID but the method grouping might not stay as orderly.

Updated Output (With Faults Live)

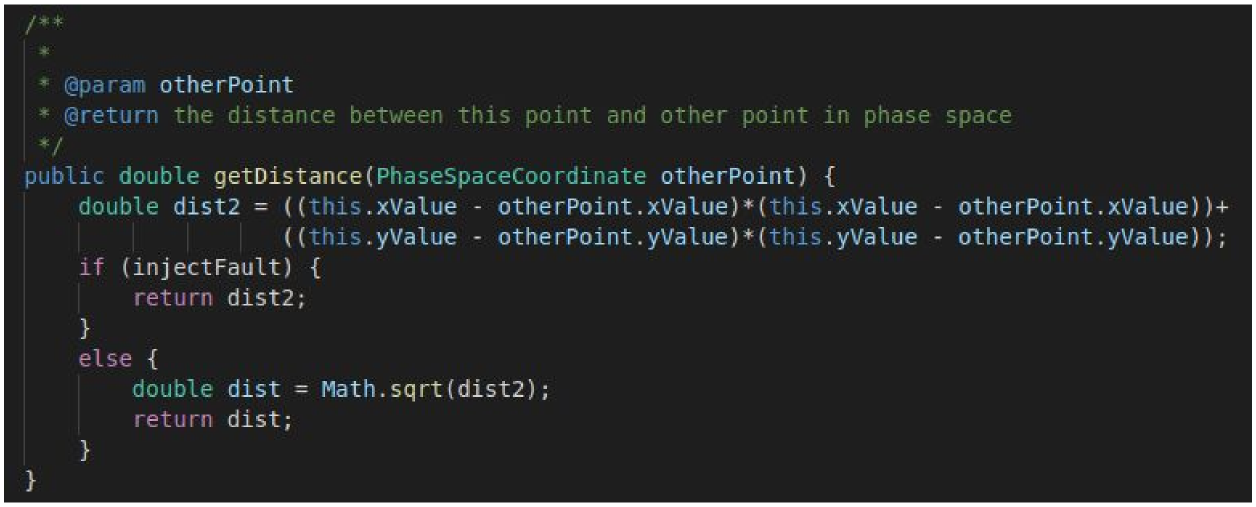


ID	Method	Requirement	Input	Oracle	Output	Result
1	calculateSlope	Method calculates the slope of a line	5 3 4 7	-4.0	-176.0	Fail
2	calculateSlope	Method calculates the slope of a line	-3 3 3 -3	-1.0	1.0	Fail
3	calculateSlope	Method calculates the slope of a line	136 -38 17 -32	-0.05042016806722689	1.5630252100840336	Fail
4	calculateSlope	Method calculates the slope of a line	35943 4037823 132894 650983	-34.93352311992656	-133.5103817126298	Fail
5	calculateSlope	Method calculates the slope of a line	64.2387634 64.5908703477 64.24089753 64.5906543	-0.10123448901101095	-7.288242889080816E9	Fail
6	compareTo	Method sorts coordinates by their x value	5 -63 72 38	-1.0	1.0	Fail
7	compareTo	Method sorts coordinates by their x value	0 0 1 0	-1.0	1.0	Fail
8	compareTo	Method sorts coordinates by their x value	136 -38 17 -32	1.0	-1.0	Fail
9	compareTo	Method sorts coordinates by their x value	1 0 0 0	1.0	-1.0	Fail
10	compareTo	Method sorts coordinates by their x value	92 92 92 92	0.0	0.0	Pass
11	formatLatLngValue	Method converts double value into a fractional string with default number of decimal places	1253404.47262174 3	1253404.472	1253404.4444444445	Fail
12	formatLatLngValue	Method converts double value into a fractional string with default number of decimal places	0.128427 4	0.1284	0.125	Fail
13	formatLatLngValue	Method converts double value into a fractional string with default number of decimal places	-126.1253799 0	-126.0	NaN	Fail
14	formatLatLngValue	Method converts double value into a fractional string with default number of decimal places	-54727143.0234885 2	-5.472714302E7	-5.4727143E7	Fail
15	formatLatLngValue	Method converts double value into a fractional string with default number of decimal places	9120370981409.12309714287894513 8	9.223372036854776E10	9.12037098140911E12	Fail
16	getDistance	Method returns the distance between this point and other point in phase space	5 5 5 5	0.0	0.0	Pass
17	getDistance	Method returns the distance between this point and other point in phase space	7 3 4 9	6.708203932499369	45.0	Fail
18	getDistance	Method returns the distance between this point and other point in phase space	-52 -3 -23 -45	51.03920062069938	2605.0	Fail
19	getDistance	Method returns the distance between this point and other point in phase space	0 0 0 0	0.0	0.0	Pass
20	getDistance	Method returns the distance between this point and other point in phase space	120983 12349078 487094 430803248	4.18454330157609E8	1.75104026427653216E17	Fail
21	lnFactorial	Method computes the log(n!)	0	0.0	0.0	Pass
22	lnFactorial	Method computes the log(n!)	-5	0.0	0.0	Pass
23	lnFactorial	Method computes the log(n!)	2000000	2.701732365031526E7	0.0	Fail
24	lnFactorial	Method computes the log(n!)	1	0.0	0.0	Pass
25	lnFactorial	Method computes the log(n!)	3	1.791759469228055	0.0	Fail

Deliverable #5: Injecting Faults

When we were first approaching the problem of needing to inject faults into the methods that we are testing that would result in some test cases passing and some failing, we had to consider what types of faults would be realistic. Ie., a person could make a mistake in computing some equation, and it would not break the code, and perhaps it might even work for some inputs, but not all of them. Thus, we scrutinized our code to come up with realistic mistakes/faults that someone could make when implementing these methods. We used a boolean variable to implement the faults, so that if it is set to true, then the faulted code will execute, otherwise, the original code will execute.

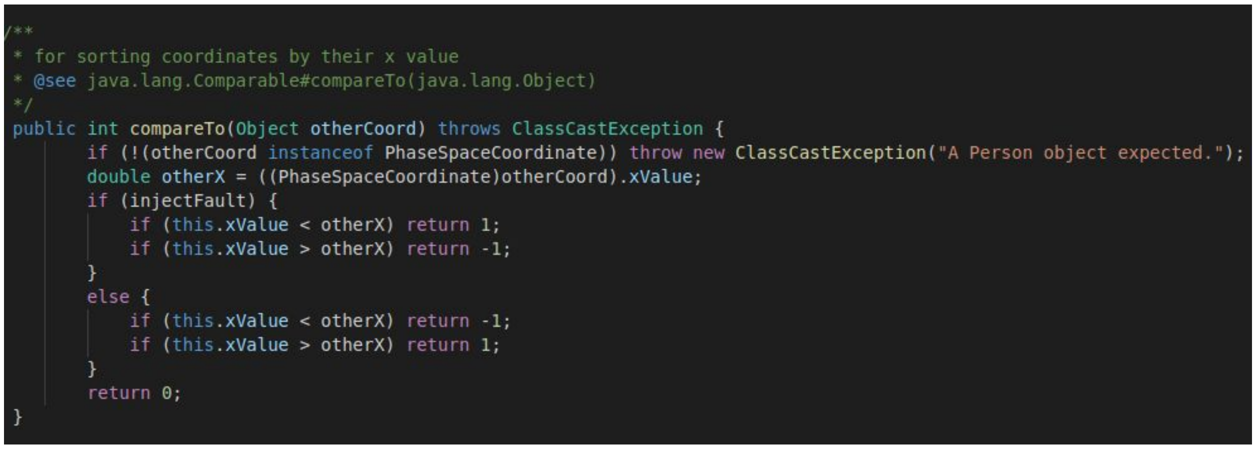
Method: getDistance



```
/**
 *
 * @param otherPoint
 * @return the distance between this point and other point in phase space
 */
public double getDistance(PhaseSpaceCoordinate otherPoint) {
    double dist2 = ((this.xValue - otherPoint.xValue)*(this.xValue - otherPoint.xValue))+
                   ((this.yValue - otherPoint.yValue)*(this.yValue - otherPoint.yValue));
    if (injectFault) {
        return dist2;
    }
    else {
        double dist = Math.sqrt(dist2);
        return dist;
    }
}
```

When considering what could go wrong with implementing the Euclidean distance formula, we thought that it might be realistic for someone to compute the ‘inside’ of the distance formula, and then after computing it they would return that value, instead of taking the square root of the computed value. This fault resulted in failing 3/5 of the test cases. The two test cases that passed were when the points were all the same, or they were all 0.

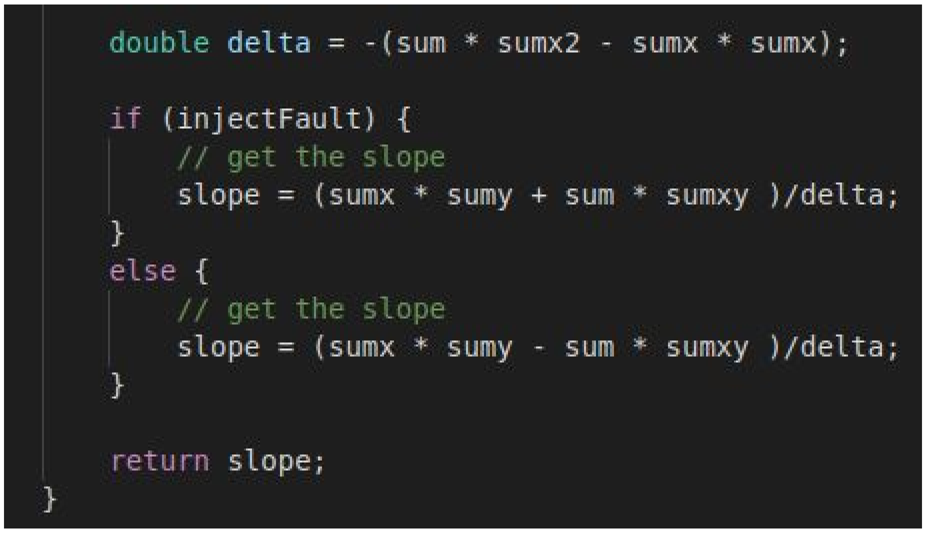
Method: compareTo



```java
/**
 * for sorting coordinates by their x value
 * @see java.lang.Comparable#compareTo(java.lang.Object)
 */
public int compareTo(Object otherCoord) throws ClassCastException {
        if (!(otherCoord instanceof PhaseSpaceCoordinate)) throw new ClassCastException("A Person object expected.");
        double otherX = ((PhaseSpaceCoordinate)otherCoord).xValue;
        if (injectFault) {
            if (this.xValue < otherX) return 1;
            if (this.xValue > otherX) return -1;
        }
        else {
            if (this.xValue < otherX) return -1;
            if (this.xValue > otherX) return 1;
        }
        return 0;
}
```

Here we decided that it could be possible for someone to accidentally switch the proper returns for the conditionals. This failed all test cases besides one, where the inputted points were all the same.

Method: calculateSlope



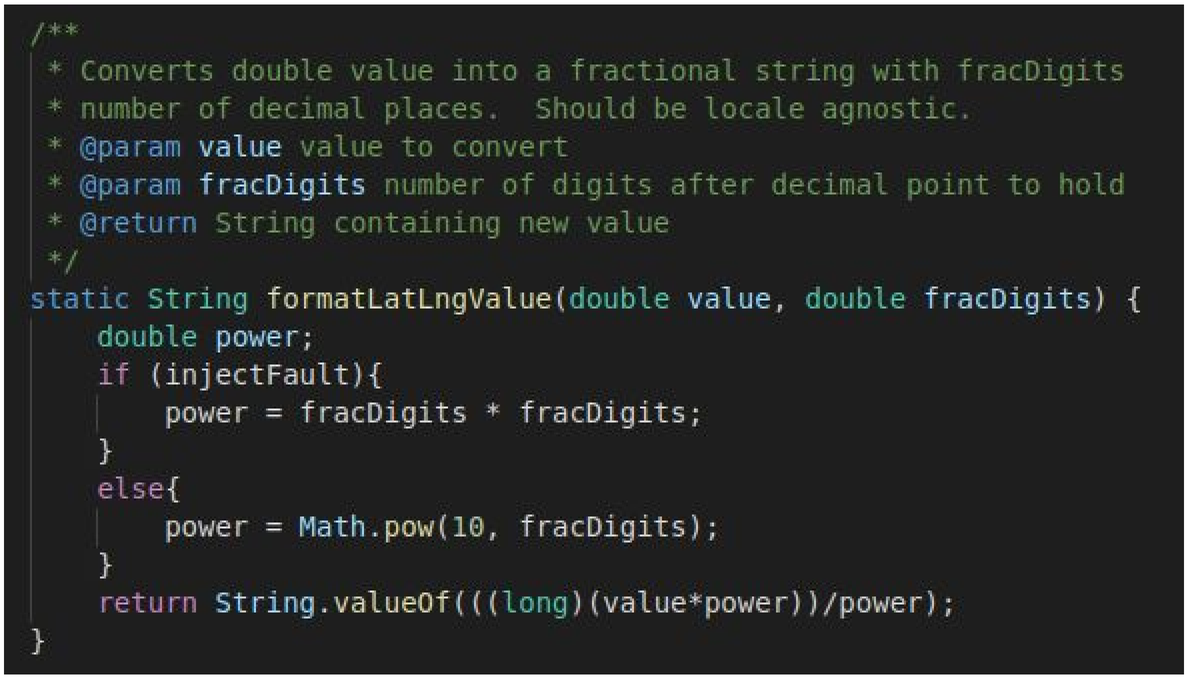
```java
        double delta = -(sum * sumx2 - sumx * sumx);

        if (injectFault) {
            // get the slope
            slope = (sumx * sumy + sum * sumxy )/delta;
        }
        else {
            // get the slope
            slope = (sumx * sumy - sum * sumxy )/delta;
        }

        return slope;
    }
```

For this fault, we used subtraction where there was supposed to be addition. It failed all test cases.

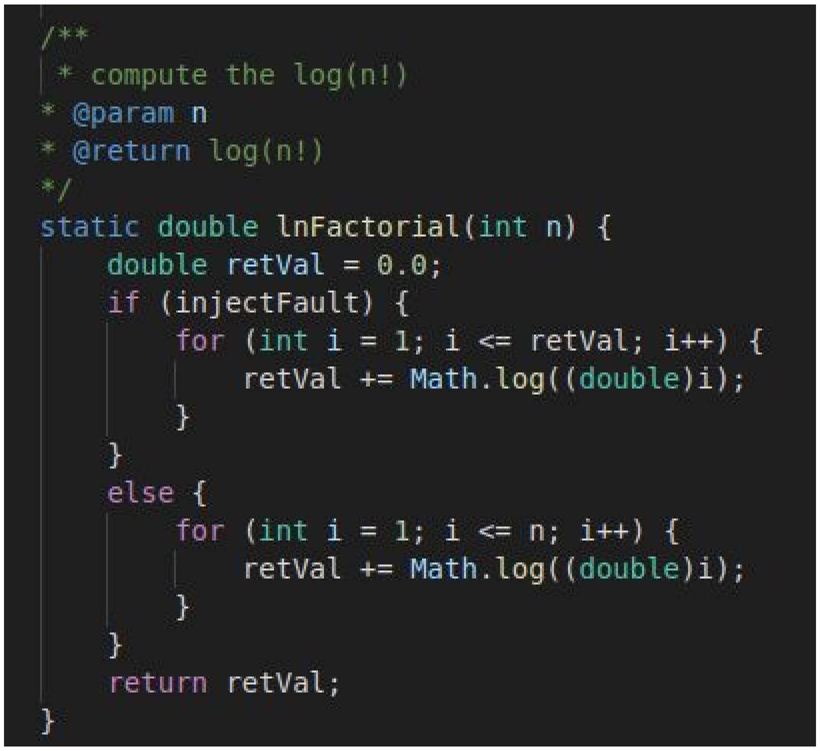
Method: formatLatLngValue



```java
/**
 * Converts double value into a fractional string with fracDigits
 * number of decimal places.  Should be locale agnostic.
 * @param value value to convert
 * @param fracDigits number of digits after decimal point to hold
 * @return String containing new value
 */
static String formatLatLngValue(double value, double fracDigits) {
    double power;
    if (injectFault){
        power = fracDigits * fracDigits;
    }
    else{
        power = Math.pow(10, fracDigits);
    }
    return String.valueOf(((long)(value*power))/power);
}
```

For this method, we couldn't think of something that would still pass some of the test cases, but we decided that someone could make a mistake when formatting the number of decimal places by computing the power as fracDigits * fracDigits, instead of taking 10^fracDigits.

Method: lnFactorial



```
/**
 * compute the log(n!)
* @param n
* @return log(n!)
*/
static double lnFactorial(int n) {
    double retVal = 0.0;
    if (injectFault) {
        for (int i = 1; i <= retVal; i++) {
            retVal += Math.log((double)i);
        }
    }
    else {
        for (int i = 1; i <= n; i++) {
            retVal += Math.log((double)i);
        }
    }
    return retVal;
}
```

For this fault, we changed how many times the for loop would iterate for computing the log(n!). We thought it might be possible for someone to make a mistake and set the for loop to run retVal times, instead of n times. This passed three test cases, and failed two. It passed the test cases where the inputted value was -5, 0, and 1.

Final Thoughts

Ultimately, we found that it was really important to fully test methods like these, because faults can exist that are not going to cause errors to be thrown, but could cause the method to behave in a way that it is not supposed to. This deliverable has been enlightening to us in the importance of double/triple/quadruple checking things, and making sure that the test cases are catching a wide range of potentialities of the method uses. Thus, shining a light on the fact that errors are not going to be caught by the compiler 100% of the time, because logic errors need to be found via exhaustive testing or more human scrutinization of the code.