Project 3 Testing Preface

This testing document was written by making use of the markdown editor *Obsidian*. This document will frequently reference my testing file *CalculatorTest.java*.

My Methodology

I worked through all of these tests making use of a variation of the Test 0, Test 1, Test many methodology outlined in the project instructions. In some cases, I tested negative values in order to ensure the effectiveness of my code. I will indicate which tests I am referring to thorugh descriptions at the start of my bulletted lists. As per the instrcutions of the project, I will not be including the actual code in this document.

I used Maven to run, compile, and Test all of the files in this project. This is a project manager for java, and it should be noted that the code compiles and tests in DrJava, as per the guidelines of this course. If for some reason you get a compiler error, I almost guarantee it is nothing wrong with my code. If you are not using a project manager, I believe removing the package declarations at the top of files and the beginning of enum imports should do the trick. I think adding my code to a package called 'projectThree' would also work. **I ask that you do not take of points for compilation errors, as everything compiles on my end and all tests run without issue.**

I used the following imports in my testing file:

```
import org.junit.*;
import org.junit.experimental.theories.suppliers.TestedOn;
import static org.junit.Assert.*;

import projectThree.BinaryOp.Op;
import projectThree.UnaryOperation.Unary;
```

Organization:

There is no actual code included in this file. The java testing file itself has javadoc comments for both JUnit methods and the private helper methods. Fields are declared in blocks and do not have indivual comments due to their similariy.

I tried my best to indicate which line tests were on, but that proved to be somewhat difficult as I made small changes to the format of the java file.

Final Note: You cannot declare null operators or operands. This is intentional. A null Function cannot exist because the operators and operands are immutable.

Fields

All fields, as mentioned above, are listed in blocks with a brief comment above them explaining where they will be used. The field's contents are all declared in helper methods, and they are often referenced or even changed in other methods in the file. Sometimes, I reference old fields by calling a helper method in a test method for testing purposes.

Testing the Actual Methods

JUnit tests will be labeled accordingly, and I will omit any private helper methods as they do not contribute to the actual testing document.

variableTest()

LINE 45

- I tested the `equals` method first. This was done by comparing two variable objects, as they should always be the same. I then checked Variable against my false assertion String to make sure the implementation can take normal Objects as well.
- Then I tested the `toString` implementation. Obviously "x" should be returned as "x" is the only String used for the variable class.
- I then tested the `value(double x)` method implementation, which should simply output the exact same double value as inputted. This was the case as confirmed by my tests that used 0 and 1 as inputs.
- Then I tested the `derivative` method by comparing the derivative returned to a new Number object with value 1. I knew that the derivative should be 1 because a linear function x will always have a derivative of 1.
- Finally, I tested the exception Handler in the class by trying to call `value()` on a variable. Variables need an input to be evaluated, so an exception was thrown as confirmed by the expected exception declaration not throwing a fit.

numberEqualsTest()

LINE 75

- I tested the getter method for the number first. I knew that I declared a Number object with value 0, so I compared that with the output of the getter method to test my method.
- I then tested the same Number object with its equals method and got true as expected.
- I tested two different Number objects with the same number being stored with the `equals` method, which returned true as expected.
- I tested a Number object against a Number object with a different double value which was false as expected.
- Finally I tested a Number object against my false assertion string which is and should be false for all tests that use it.

numberMethodTest()

LINE 88

- I tested the `toString` method first, using two Number objects that both had different double values stored in them. numOne stores 0, so `toString` returned "0.0" as expected. numThree stores 1, so toString returned "1.0" as expected.

- I then tested the `value()` and `value(double x)` implementations for the objects referenced in the previous bullet point. Regardless of the arguments, or lack thereof, in the methods, the output should be the same as the value stored in the objects. This was tested using no input, an input, and a random input. All tests with these parameters passed.
- Finally, I tested the `derivative` method. The derivative of any Number is 0 by definition. This was confirmed by the 3 tests as the derivative being 1 was false, but the derivative being 0 for two different numbers was true.

binaryNullTest()

LINE 141

- I tested the implementation of my `AbstractFunction` constructor that refuses to accept null as an input for any operator. For BinaryOp, I tested the null operator in all 3 possible positions, followed by the null operator in all three positions at the same time. I set the constructor to throw a `NullPointerException`, as confirmed by this test.

binaryOperatorTest()

LINE 153

- First I tested the getter methods for all 4 possible operators. I compared the Op Object itself to the BinaryOp objects themselves after using the `getOperator` method. The objects were all equal with respect to their operators showing a successful test.
- I then tested the `getLeftOperand` and `getRightOperand` methods for the BinaryOp type. I used two different objects for this for robustness. The operands were equal to direct creations of the Objects stored in the operands, proving that the methods work properly.

binaryEqualsTest()

LINE 170

- First I tested my non-function object with the `equals` method for BinaryOp. This should be and was false by design.

- I tested various BinaryOp objects against a variety of Objects and BinaryOps of which I knew were not equal. This covered the multiple if statements in the BinaryOp class that are designed to return false is any of the tests I did are false. If the Operator, leftOperand, or rightOperand are unequal in any way, then false is returned by default. If all of those conditions pass, true is returned as seen with the final assertion in this method.

binaryToStringTest()

LINE 206

- I first tested simple Binary Operations where the are two arguments and an operator. These are formatted with a space between each component as confirmed by the tests. This bypasses conditionals in the method.
- Then I tested a nested BinaryOp that is of the same operator type. Since the operators are equal, no parentheses are appended and the BinaryOp test passes as expected.
- Then I tested the possibility of a right operand being a BinaryOp with a different operator than the parent BinaryOp. Because of this, the `toString` method appends parentheses around the right operator.
- Then I tested the possibility of a left operator being a BinaryOp. The method should append parentheses regardless of the operator, as confirmed by the two tests in this JUnit method.
- Finally I tested the possibility of two nested BinaryOp in the parent BinaryOp. All of them have different Operators, so everything except the parent operator in in paratheses as per the project instructions.

binaryNumericTest()

LINE 225

- I tested numeric only BinaryOps, meaning that Number objects were the only Functions nested in the BinaryOps tested here. This caused the `value` method to return the exact value of the BinaryOp consistently, as confirmed by the tests.

binaryValueTest()

LINE 244

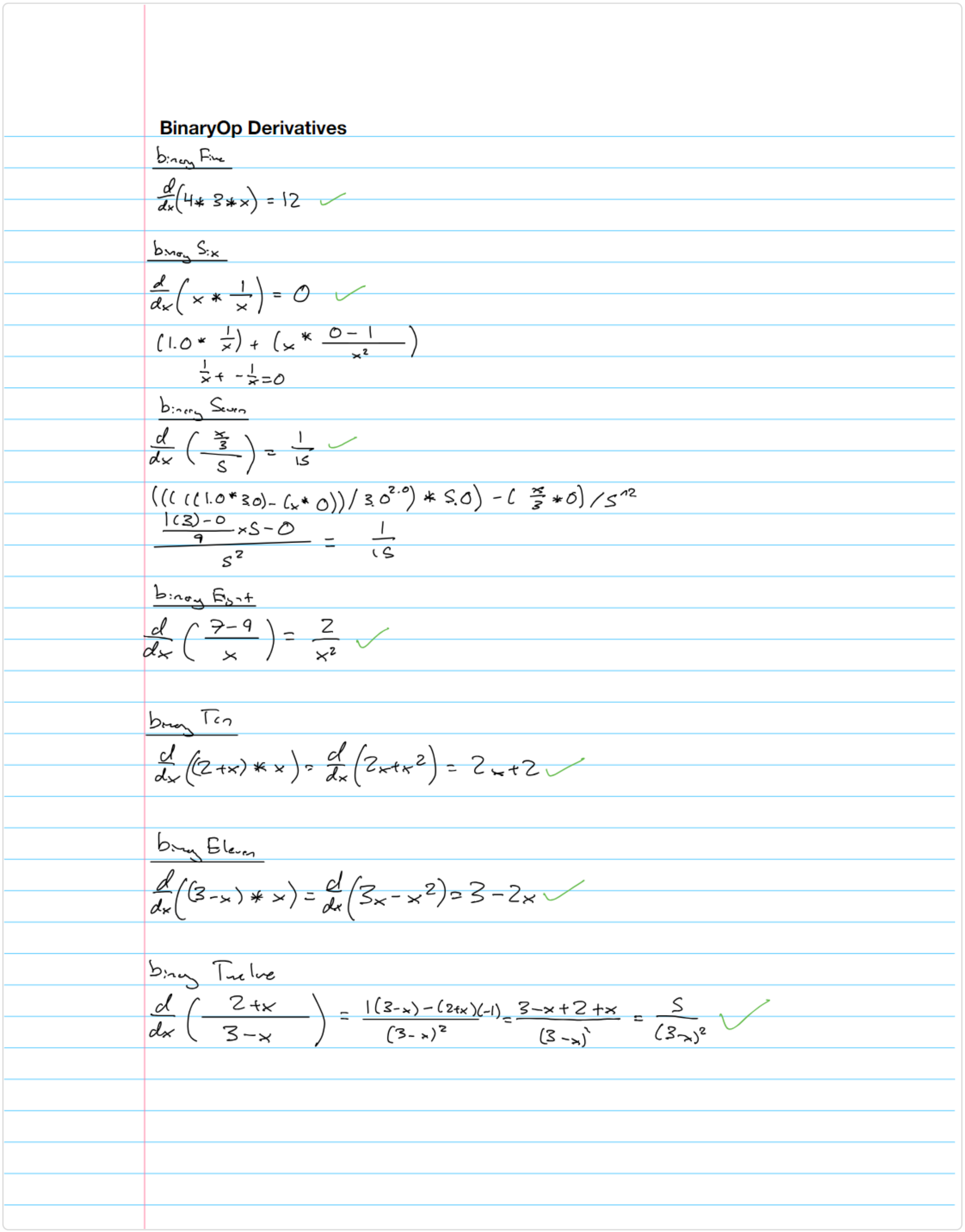
- I then tested complex BinaryOps that contained Variables and nested BinaryOps. These depended on input from `value`, and throw an exception when no input is provided. Because of my stated expected exception and use of a calcultor to evaluate the BinaryOps at my chosen values, I concluded that my `value` implementation was valid in this class for both having and not having an input.

binaryDerivativeTest()

LINE 267

These tests are messy, I apologize!

- Since the implementation of the `derivative` method here is crutial here for the rest of the tests, I wrote a ton of tests. The switch statement in BinaryOp handles the four different derivative rules.
- For the first 4 tests, I painstakingly defined the type of the derivative by hand, which I will not do for the rest of this report. Calling the `derivative` method on the given BinaryOps returned the expected Function Objects, confirming the success of these tests.
- Then I started comparing derivatives using `toString`. This saved me a lot of time, and shows the actual way BinaryOp takes derivatives. Almost everything is a product rule or quotient rule. This prematurely used the Polynomial class, but I thought that was ok because it clearly worked as intended. Although the derivatives look very complicated, they typically simplify down to short expressions. This calculator does not simplify, so I wrote out the derivatives by hand. I confirmed them by hand as seen below and then with the method.



BinaryOp Derivatives

binary Five

$\frac{d}{dx}(4 * 3 * x) = 12$ ✓

binary Six

$\frac{d}{dx}\left(x * \frac{1}{x}\right) = 0$ ✓

$(1.0 * \frac{1}{x}) + \left(x * \frac{0-1}{x^2}\right)$

$\frac{1}{x} + -\frac{1}{x} = 0$

binary Seven

$\frac{d}{dx}\left(\frac{\frac{x}{3}}{5}\right) = \frac{1}{15}$ ✓

$(((((1.0 * 3.0) - (x * 0))/3.0^{2.0}) * 5.0) - (\frac{x}{3} * 0)/5^{\wedge 2}$

$\frac{\frac{1(3)-0}{9} \times 5 - 0}{5^2} = \frac{1}{15}$

binary Eight

$\frac{d}{dx}\left(\frac{7-9}{x}\right) = \frac{2}{x^2}$ ✓

binary Ten

$\frac{d}{dx}((2+x) * x) = \frac{d}{dx}(2x + x^2) = 2x + 2$ ✓

binary Eleven

$\frac{d}{dx}((3-x) * x) = \frac{d}{dx}(3x - x^2) = 3 - 2x$ ✓

binary Twelve

$\frac{d}{dx}\left(\frac{2+x}{3-x}\right) = \frac{1(3-x)-(2+x)(-1)}{(3-x)^2} = \frac{3-x+2+x}{(3-x)^2} = \frac{5}{(3-x)^2}$ ✓

It should be noted that I will not be writing out any other derivatives by hand! All other derivative checks are done with Wolfram Alpha, as it is much faster than I am at taking complex derivatives.

polynomialNullTest()

LINE 321

- This method exists to test the `AbstractFunction` Constructor usage. Trying to create a Polynomial with a null Function leads to a NullPointerException, as confirmed by this method.

polynomialGettersTest()

LINE 329

- I interpreted 0, 1, Many as numeric representations inside the Polynomial Object. I tested the getter methods for the Operator and the power for various different objects with the previous interpretation. All of the tests passed making use of the Variable and Number implementation of the `equals` method and comparisons of double values against the powers stored in the Polynomial objects.

polynomialEqualsTest()

LINE 371

- I first tested 0, which I let be a polynomial that only had 0 in its arguments. The two objects were equal.
- Then I tested 1, which was a polynomial with 1 as both its arguments. These two objects were also equal.
- Then I tested a Polynomial against my String false asserter, which was not equal by design.
- I then did a series of 'Test Many' by using the `equals` method of the Polynomial class with Polynomials that complex nested types. All of the varieties of these tests passed, showing that the method works even when it has to look many levels through a Function object.

polynomialToStringTest()

LINE 389

- I first tested a Polynomial with all 0s, the string should be 0.0^0.0, as confirmed here.
- Then I tested the same thing, but with all 1s. This was confirmed to work as well.

- Then I tested an incorrect string, the 'neqTester', against the `toString` call on a Polynomial. Although both objects are Strings, their contents are not equal.
- I then went overboard with the test many. I covered the BinaryOp conditional, which correctly placed parentheses around BinaryOp arguments nested inside Polynomials. The rest of the tests may seem redundant, but I wanted to ensure the rigorousness of my method. All tests passed regardless of the complexity of the Polynomial, nested or not.

polynomialValueTest()

LINE 412

- I first tested 0^0, with various arguments in `value`. All of the outputs were 1, because java evaluates 0^0 to 1 and my method ignores input if the operator does not contain Variables. I did the same tests on 2^2^2, which all evaluated to 16 as expected.
- I then tested a variety of Polynomials that all evaluated to various values. I confirmed the results using my calculator and by following the operation order specified in the `toString` result of the corresponding Polynomial.
- I then tested a fractional power with a negative argument. Since my calculator does not account for complex numbers, I used the Double classes `isNaN` method to confirm the result of this test.
- Finally, I tested the exception handler when there was not an input given for a Polynomial that requires one to be evaluated.

polynomialDerivativesTest()

LINE 443

These tests are messy, I apologize!

- I first tested 0 by evaluating a Polynomial with a Number of 0 value in the Function position. Because of the implementation of `derivative` that has a conditional to handle an argument of 0, "0.0" is returned when the derivative is converted to a string.
- Then I tested 1 by evaluating a Polynomial with 1 in each position. This triggers another conditional where 1 becomes the first term. Because the inner Function is a number, 0 is returned which is confirmed by the formatted string representation of the derivative taken in this assertion.

- Then I tested very long expressions. These expression were very difficult to derive, so I computed them with Wolfram Alpha and compared them with the output given by the method. All of the outputs from Wolfram aligned with the calculated derivative using the method, as confirmed by the tests in this method.
- polyEleven was used to show that actual polynomial expressions, like the ones you'd find in a math class, can be made and their derivatives can be computed as well.

unaryTest()

LINE 491

You will notice that there are 4 blocks of code here, all looking almost exactly the same, minus the change in operator. I will summarize this test briefly

- First I fetched the operator of the expression and compared it to an explicit definition of the operator. This tested and confirmed the `equals` method in the Unary nested Type.
- Then I tested the `getOperand` method for the UnaryOperator type. All 4 types worked perfectly for this, checking off the requirement for a `getOperand` method for Log, Exp, Sin, and Cos.
- Then I tested the `getUnaryName` method which is crucial for the `toString` method for subtypes. The 4 operators were returned correctly as Strings with the first letter capitalized as per the project instructions.
- Finally, I tested a false assertion against the Operator of the Type. This shows that the `equals` method works properly and correctly asserts when an Object is equal to an Operator or not.

Repetitive

The following tests are very repetitive. They all test similar things as they are all part of the parent type `UnaryOperation`. All tests are explained, though similar language will be consistently used.

nullLogTest()

LINE 546

- This test exists to ensure the supertype constructor works when you try to put null as an Operand for a Log operation.

logEqualsTest()

LINE 555

- I first explicitly compared two Log objects with their exact declarations. Both of these tests were true, as they should be.
- Then, I tested a false Assertion with my 'neqTester' used earlier. This was false by design.
- I then tested log expressions that were completely different from one another. One test bypassed the conditional in the super class, comparing the operands only. The other triggered the conditional to return false immediately because the types were not equal. Since they weren't equal, false was returned as expected.
- Then I tested Log expressions that had nested Polynomials and BinaryOps. Since these tests were true and false respectively, it showed the robustness of the Log `equals` branch when facing nested types.
- Finally, I tested the individual operators in the last Log expression, which was actually a BinaryOp. This compared the complex nested Log expressions, which were equal, further showing that the `equals` method works well.

logToStringTest()

LINE 580

- I first tested constants inside the log expression. These tests worked as expected, as the `toString` method places the double values inside the square brackets as required by the project.
- I then tested a Variable inside a Log expression, which worked as expected. The result adheres to the projects guidelines, and the test passed successfully.
- I then tested nested Polynomials, which passed their tests as well with proper formatting.
- Then, I tested a BinaryOp inside a Log Expression that composed of a Log expression. This tested the delimiter matching, which was correct and matches the projects expectations.
- Finally, I tested the sum of two Log expressions, with the second expression have a nested Log Expression inside a polynomial. This further tested delimiter matching, which was correct as confirmed by the assertion.

logValueTest()

LINE 597

- I First tested undefined expressions for Log. Since the log of any negative number is complex, java returns NaN, which I confirmed with the Double classes `isNaN` method. I did something similar for the log of 0, but java interprets this expression as ∞, so I used the Double classes `isInfinite` method to check the equality. These tests showed that `value()` and `value(double x)` work for the log class even when undefined.
- I then tested various inputs in the `value(double x)` method, and confirmed them using my calculator. I realized that java calculates the expression to more accuracy than my Ti-84, so I had to use a margin of error of 0.0001 when necessary. The equivalence of the expressions within the margin of error shows that the method works.
- Finally, I showed that my Exception handler works for the Log class with the attempt of calling `value()` on a variable Log expression.

logDerivativeTest()

LINE 620

- Even though Log of a negative number is imaginary, its derivative is still zero as confirmed by my first test.
- Then I tested the 0^0 function which java interprets as infinite. This result can also be interpreted as indeterminate, which is confirmed by the derivative of $\frac{0}{0}$.
- Then I calculated the derivative of a constant expression which was 0 as expected.
- I then tested a variety of derivatives with nested expressions. The first derivative I tested in this block confirmed the base behavior of $\frac{dy}{dx}(Log[x])$. This was correct so I proceeded to calculate derivatives of longer expressions that I confirmed with Wolfram Alpha.

nullExpTest()

LINE 662

- This test exists to ensure the supertype constructor works when you try to put null as an Operand for a Exp operation.

expEqualsTest()

LINE 671

- I first explicitly compared two Exp objects with their exact declarations. Both of these tests were true, as they should be.
- Then, I tested a false Assertion with my 'neqTester' used earlier. This was false by design.
- I then tested Exp expressions that were completely different from one another. One test bypassed the conditional in the super class, comparing the operands only. The other triggered the conditional to return false immediately because the types were not equal. Since they weren't equal, false was returned as expected.
- Then I tested Exp expressions that had nested Polynomials and BinaryOps. Since these tests were true and false respectively, it showed the robustness of the Exp `equals` branch when facing nested types.
- Finally, I tested the individual operators in the last Exp expression, which was actually a BinaryOp. This compared the complex nested Exp expressions, which were equal, further showing that the `equals` method works well.

expToStringTest()

LINE 697

- I first tested constants inside the Exp expression. These tests worked as expected, as the `toString` method places the double values inside the square brackets as required by the project.
- I then tested a Variable inside a Exp expression, which worked as expected. The result adheres to the projects guidelines, and the test passed successfully.
- I then tested nested Polynomials, which passed their tests as well with proper formatting.
- Then, I tested a BinaryOp inside a Exp Expression that composed of a Log expression. This tested the delimiter matching, which was correct and matches the projects expectations.
- Finally, I tested the sum of two Log expressions, with the second expression have a nested Log Expression inside a polynomial. This further tested delimiter matching, which was correct as confirmed by the assertion.

expValueTest()

LINE 713

- I first tested various inputs in the `value(double x)` method, and confirmed them using my calculator. I used a margin of error of 0.0001 when necessary. The equivalence of the expressions within the margin of error shows that the method works.
- Finally, I showed that my Exception handler works for the Exp class with the attempt of calling `value()` on a variable Exp expression.

expDerivativeTest()

LINE 737

- First I calculated derivatives of constant expressions. All of these expressions were presented in correct BinaryOp format, and all simplify to 0 as expected.
- I then tested a variety of derivatives with nested expressions. The first derivative I tested in this block confirmed the base behavior of $\frac{dy}{dx}(Exp[x])$. This was correct so I proceeded to calculate derivatives of longer expressions that I confirmed with Wolfram Alpha.

nullSinTest()

LINE 779

-This test exists to ensure the supertype constructor works when you try to put null as an Operand for a Sin operation.

sinEqualsTest()

LINE 788

- I first explicitly compared two Sin objects with their exact declarations. Both of these tests were true, as they should be.
- Then, I tested a false Assertion with my 'neqTester' used earlier. This was false by design.
- I then tested Sin expressions that were completely different from one another. One test bypassed the conditional in the super class, comparing the operands only. The other triggered

the conditional to return false immediately because the types were not equal. Since they weren't equal, false was returned as expected.

- Then I tested Sin expressions that had nested Polynomials and BinaryOps. Since these tests were true and false respectively, it showed the robustness of the Sin `equals` branch when facing nested types.
- Finally, I tested the individual operators in the last Sin expression, which was actually a BinaryOp. This compared the complex nested Sin expressions, which were equal, further showing that the `equals` method works well.

sinToStringTest()

LINE 814

- I first tested constants inside the Sin expression. These tests worked as expected, as the `toString` method places the double values inside the square brackets as required by the project.
- I then tested a Variable inside a Sin expression, which worked as expected. The result adheres to the projects guidelines, and the test passed successfully.
- I then tested nested Polynomials, which passed their tests as well with proper formatting.
- Then, I tested a BinaryOp inside a Sin Expression that composed of a Sin expression. This tested the delimiter matching, which was correct and matches the projects expectations.
- Finally, I tested the sum of two Sin expressions, with the second expression have a nested Sin Expression inside a polynomial. This further tested delimiter matching, which was correct as confirmed by the assertion.

sinValueTest()

LINE 829

- I first tested various inputs in the `value(double x)` method, and confirmed them using my calculator. I used a margin of error of 0.0001 when necessary. The equivalence of the expressions within the margin of error shows that the method works.
- Finally, I showed that my Exception handler works for the Cos class with the attempt of calling `value()` on a variable Cos expression.

sinDerivativeTest()

LINE 851

- First I calculated derivatives of constant expressions. All of these expressions were presented in correct BinaryOp format, and both simplified to 0 as expected.
- I then tested a variety of derivatives with nested expressions. The first derivative I tested in this block confirmed the base behavior of $\frac{dy}{dx}(Sin[x])$. This was correct so I proceeded to calculate derivatives of longer expressions that I confirmed with Wolfram Alpha.

nullCosTest()

LINE 892

- This test exists to ensure the supertype constructor works when you try to put null as an Operand for a Cos operation.

cosEqualsTest()

LINE 901

- I first explicitly compared two Cos objects with their exact declarations. Both of these tests were true, as they should be.
- Then, I tested a false Assertion with my 'neqTester' used earlier. This was false by design.
- I then tested Cos expressions that were completely different from one another. One test bypassed the conditional in the super class, comparing the operands only. The other triggered the conditional to return false immediately because the types were not equal. Since they weren't equal, false was returned as expected.
- Then I tested Cos expressions that had nested Polynomials and BinaryOps. Since these tests were true and false respectively, it showed the robustness of the Cos `equals` branch when facing nested types.
- Finally, I tested the individual operators in the last Cos expression, which was actually a BinaryOp. This compared the complex nested Cos expressions, which were equal, further showing that the `equals` method works well.

cosToStringTest()

LINE 926

- I first tested constants inside the Cos expression. These tests worked as expected, as the `toString` method places the double values inside the square brackets as required by the project.
- I then tested a Variable inside a Cos expression, which worked as expected. The result adheres to the projects guidelines, and the test passed successfully.
- I then tested nested Polynomials, which passed their tests as well with proper formatting.
- Then, I tested a BinaryOp inside a Cos Expression that composed of a Cos expression. This tested the delimiter matching, which was correct and matches the projects expectations.
- Finally, I tested the sum of two Cos expressions, with the second expression have a nested Cos Expression inside a polynomial. This further tested delimiter matching, which was correct as confirmed by the assertion.

cosValueTest()

LINE 942

- I then tested various inputs in the `value(double x)` method, and confirmed them using my calculator. I used a margin of error of 0.0001 when necessary. The equivalence of the expressions within the margin of error shows that the method works.
- Finally, I showed that my Exception handler works for the Cos class with the attempt of calling `value()` on a variable Cos expression.

cosDerivativeTest()

LINE 964

- First I calculated derivatives of constant expressions. All of these expressions were presented in correct BinaryOp format, and both simplified to 0 as expected.
- I then tested a variety of derivatives with nested expressions. The first derivative I tested in this block confirmed the base behavior of $\frac{dy}{dx}(Cos[x])$. This was correct so I proceeded to calculate derivatives of longer expressions that I confirmed with Wolfram Alpha.

> ***That concludes the testing portion of this project***
>
> ```
> I appologize for funky formatting and length
> ```

Spell check does not work all the time in code blocks, so I apologize for minor errors or copy-paste issues, as some of the tests were very repetitive.