Exercise 1

(a) Write the following interfaces[1] :

- An interface `EnhancedValue` which has all the methods of the interface `Value` you have seen during the lecture (with the difference that now these methods will return an object of type `EnhancedValue` and will take as input objects of type `EnhancedValue`) with two more methods, i.e., `EnhancedValue exp()` and `EnhancedValue addProduct(EnhancedValue x, EnhancedValue y)`.
- An interface `EnhancedValueDifferentiable` extending `EnhancedValue`, with a method `EnhancedValue getDerivativeWithRespectTo(EnhancedValueDifferentiable x)`.
- An interface `ConvertableToFloatingPoint` with a method `Double asFloatingPoint()`.

(b) Write a class `EnhancedValueDoubleDifferentiable` implementing the interfaces `EnhancedValueDifferentiable` and `ConvertableToFloatingPoint`. This will be similar to the class `ValueDoubleDifferentiable` seen in the lecture, with the improvement that you now provide the implementation and the derivatives of the exponential and of the function $f(x, y, z) = x + yz$, that is represented by `addProduct` [2].

In particular, you have to provide the implementation of the methods `EnhancedValue exp()` and `EnhancedValue addProduct(EnhancedValue x, EnhancedValue y)`, in the same way it has been done during the lecture for the other operations (also adding the new operators to the `enum` field). You also have to add two cases to the `switch` statement of the method `propagateDerivativeToArguments`, according to these new operators. Here you then have to implement the code to return the right derivatives. Make sure you change all the parts of the code in a way that it now works with the classes and interfaces you have created.

Exercise 2

Test the implementation of Exercise 1 in a `JUnit test case`, in a similar way you have seen during the lecture in `ValueDoubleDifferentiableTest`. You can test the partial derivatives of the function

$$f(a, b) = e^{a^2 + a \cdot b^2},$$

that you construct using both the methods `addProduct` (where you add the product $a \cdot b^2$ to a^2) and `exp`. Alternatively, if you were not able to implement `addProduct` and its derivation (it's a bit more tricky because it has two arguments) you can just use the new method `exp` and compute $a^2 + ab^2$ just using `add` and `mult`.
Also test the derivative of the function

$$g(x) = 7 \cdot x + 7 \cdot x \cdot 4,$$

that you construct by creating a node x, a node $x_1 = 7 \cdot x$, a node $x_2 = 4$, and then letting x_1 call `addProduct` with arguments x_1 and x_2.

Exercise 3

Look at the current implementation of `net.finmath.aadexperiments.value.ValueDoubleDifferentiable`,

[1]Here you will basically copy the implementation you have seen in the lecture, adding some more functionality. It would be better to do it in an object oriented way by using composition and inheritance, but we do it in this way to keep the exercises as self contained as possible, avoiding a dependence on one more project. In general, keep in my mind to use composition and inheritance instead of just copying already implemented code.

[2]Again, for this exercise you can copy the code of `ValueDoubleDifferentiable` you need, even if in general it's not the best practice.

in the more updated version of the project `computational-finance-algorithmicdifferentiation`. Look at the code at lines 153-154, i.e., at the implementation of the case `div` in the `switch` statement of the method `propagateDerivativeToArguments`. Suppose to change those two lines with the following implementation:

```
double x = arguments.get(0).asFloatingPoint();
double y = arguments.get(1).asFloatingPoint();
double derivativeOfCurrentNode = derivatives.get(node);
double derivativeOfFirstArgumentNode = derivatives.getOrDefault(arguments.get(0),0.0);
double derivativeOfSecondArgumentNode =
derivatives.getOrDefault(arguments.get(1),0.0);
derivativeOfFirstArgumentNode = derivativeOfFirstArgumentNode +
derivativeOfCurrentNode * 1/y;
derivativeOfSecondArgumentNode = derivativeOfSecondArgumentNode -
derivativeOfCurrentNode * x/(y*y);
derivatives.put(arguments.get(0), derivativeOfFirstArgumentNode);
derivatives.put(arguments.get(1), derivativeOfSecondArgumentNode);
```

When tested by

```
final ValueDifferentiable x1 = new ValueDoubleDifferentiable(2.0);
final Value y = x1.div(x1);
final Value derivativeAlgorithmic =
((ValueDifferentiable)y).getDerivativeWithRespectTo(x1);
```

this would not give the result we expect, you can also try to do yourself the experiment. Can you guess why this happens?