Object Orientation – Spring 2018 **Assignment 5**

Goals

After making this exercise you should be able to:

- define a class hierarchy;
- create subclasses of (abstract) classes;
- use inheritance in Java;
- define attributes and methods at the right place in the hierarchy;
- make classes and methods `abstract` when this is required;
- implement and redefine methods.

Numerical Expressions

In this assignment you will make an object-oriented data structure for numerical expressions. Since this assignment is about classes, inheritance and class hierarchies you have to use a special purpose class for each kind of expressions.

Kind of Expressions

The following expressions are handled:

- Expression without arguments:
  - Constant expressions containing a `double` value;
  - Variables containing the name of the variable as a `String`;
- Single argument expressions:
  - Negate: changes the sign of a number in the evaluation;
  - *optionally* you can add expressions like square root;
- Double argument expressions
  - Addition: sums the values of both subexpressions during evaluation;
  - Multiplication yields the product of the arguments during evaluation;
  - *optionally* you can add operations like difference, division, and a power function.

Manipulations of Expressions

There are three manipulations of expression in this assignment. It is of course allowed to add additional methods that are needed or convenient in your program.

Conversion to String

Each and every expressions should have a tailored `toString` method yielding the infix representation of the expression. Typical examples are: `3.14`, `x`, and `(3+4)*y`. It is compulsory to write enough parenthesis to avoid erroneous interpretations of the expression. Avoid parenthesis for expressions without arguments. It is fine to include parenthesis that are superfluous in the common mathematical notation, like `2+(5*8)` for $2 + 5 \times 8$.

Evaluation

Expressions can be evaluated by `eval`. This method has a store, a mapping from variable names to values, as a argument and yields a value. Stores are often used in programs and hence Java offers a reusable solution[1]. The store is most easily implemented as a Java mapping from names to values:

```
    Map<String, Double> store = new HashMap<>();
    store.put("pi", 3.1415);
    store.put("a", 42.);
```

Your program can lookup the value of an identifier by `store.get("pi")`. Fill the store with name value pairs of your own choice. It is fine to assume that all occurring names are defined in the store. Hence, it is not required to add proper error handling for undefined variables.

Optimization

The final manipulation simplifies expressions by evaluating parts of the expression that do not depend on variables. Such an optimization should turn an expression like $(3 + (2 * 2)) * x$ in $7 * x$. The manipulations is known as partial evaluation or constant folding. Implement at least the following rules where n and m are numbers, and `x` and `y` are arbitrary expressions.

$$\begin{array}{rcl}
n + m & \to & n + m \\
x + 0 & \to & x \\
0 + y & \to & y \\
n \times m & \to & o \text{ where } o = n \times m \\
0 \times y & \to & 0 \\
1 \times y & \to & y \\
x \times 0 & \to & 0 \\
x \times 1 & \to & x \\
\texttt{neg(}\, n \,\texttt{)} & \to & -n
\end{array}$$

Expression Factory

The definition of expressions in Java becomes rather verbose. For instance:

```
e1 = new Add(new Multiply(new Constant(2), new Constant(3)), new Variable("x"));
```

To solve this problem, it is convenient to introduce methods that will produce the desired object. To make this reusable we collect these methods in a factory class.

A static import of this class enables us to use the `public static` members of this class without specifying the name of this class. For instance by the import

```
import static java.lang.Math.*;
```

we can use any static member of this class without writing `Math`, e.g. `Double d = max(7, PI);` instead of `Double d = Math.max(7, Math.PI);`.

This should allow us to write the expression above as:

```
e1 = add(mul(con(2), con(3)), var("x"));
```

[1]Next week we will explain the meaning of `<String, Double>`, for the moment just copy this code and enjoy. When you do not like to use things you do not fully understand you are most welcome to implement a store with methods `put(string name, double value)` and `double get(String name)` yourself.

Test cases

Write enough test cases and print their result such that the correctness of printing, evaluation and optimization by constant folding is plausible.

[Optional] Parsing Expressions

This part is **not** required. To test your program interactively it is convenient to convert a line entered by the user to an expression for your program. This expression is then printed, evaluated and optimized. The optimized version is also printed and evaluated.

An example of a valid input for this program is `add(mul(2, 3), x)`. When you want to select the various parts of this input with the Java `Scanner` you need to tell it that parenthesis and commas are separators between tokens that you want to get. This can be achieved by setting the delimiters of tokens:

```
    Scanner scan = new Scanner(line);
    scan.useDelimiter("\\s+|(?=\\()|(?=\\))|(?=,)|(?<=\\()|(?<=\\))|(?<=,))");
```

By default only white space in the input is recognized as separation between tokens.

Deliverables

Create the required classes and tests. Avoid repeating code by defining methods at the right place in the class hierarchy. Make methods you cannot yet define abstract. Make classes that should not be instantiated abstract. Note that parsing expressions is optional.

It is very helpful to draw a class diagram before you start coding, but it is not required to hand in this diagram. It is convenient to construct the parts of this diagram incrementally and to test the finished parts as early as possible (i.e. long before your entire program is completely available as Java code).

Do not forget to apply the provided layout rules. Include the JavaDoc specifying your names and student numbers.

Hand in all `.java` files from the package via Blackboard. **Deadline: Sunday March 11, 23:59**.