

Assignment Expressions

Object Orientation

Spring 2025

1 Goals

After making this exercise you should be able to:

- Define a class hierarchy with interfaces and classes
- Create subclasses of classes and 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

2 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 should use a special purpose class for every kind of expression.

2.1 Kinds of Expressions

The following expressions should be supported.

- Expressions without arguments
 - Constant expressions containing a **double** value
 - Variables
- Single argument expressions
 - Negate: changes the sign of a number in the evaluation
 - **Optional:** Other unary 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
 - **Optional:** Operators for difference, division, and power

Your task is to create a class hierarchy with interfaces, abstract classes, and classes to reflect this structure. Figure 1 shows a hint how the class hierarchy should be structured. Your task is to figure out which classes should be abstract, and where to put method implementations to avoid code duplication.

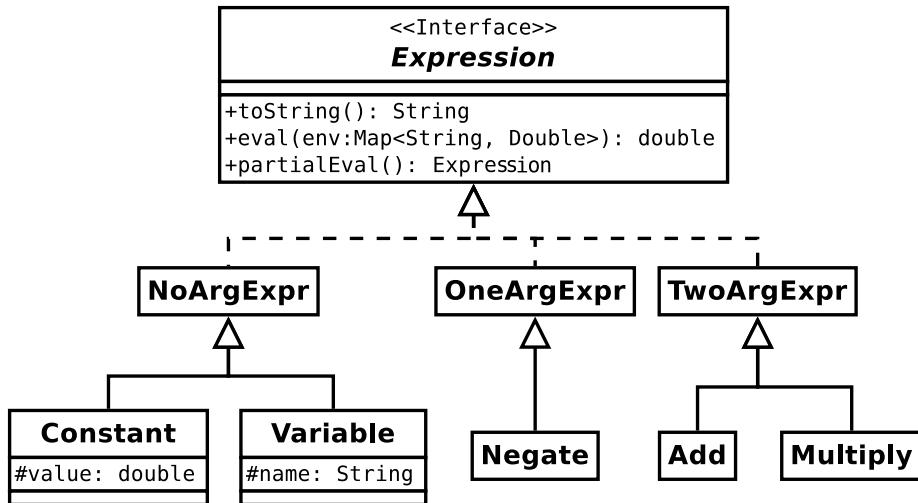


Figure 1: Hint for a class hierarchy for this assignment.

2.2 Manipulations of Expressions

Expressions can be manipulated in three ways: conversion to string, evaluation, and partial evaluation.

2.2.1 Conversion to String

Expressions should have a `toString` method yielding its infix representation. Typical examples are: `3.14` for a constant, `x` for a variable, and `(3+4)*y` for a compound expression. There should be enough parenthesis to avoid erroneous interpretations of the expression. Constants should not be surrounded by parenthesis. It is fine to include parenthesis that are superfluous in the common mathematical interpretation. For example, your program should print `2+(5*8)` for `2 + 5 × 8`.

2.2.2 Evaluation

Expressions can be evaluated by `eval`. This method needs a store as parameter, which is a mapping from variable names to values. Stores are often used in programs and hence Java offers a reusable solution. The store is most easily implemented using a `Map<String, Double>`. `Map` is part of the Java standard library¹.

Fill the store with name-value pairs of your own choice. Those are the variables that can occur in your expressions. It is not required to add proper error handling for undefined variables.

2.2.3 Constant Folding

Expressions can be simplified by evaluating all parts that do not depend on variables. For example, the partial evaluation of `(3+(2*2))*x` is `7*x`. This manipulation is known as *partial evaluation* or *constant folding*.

¹<https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java.util/HashMap.html>

The method `partialEval` should return a copy of the expression that is simplified according to the following rules. n and m are constants, and e are arbitrary expressions.

$n + m$	\rightarrow	o where $o = n + m$
$e + 0$	\rightarrow	e
$0 + e$	\rightarrow	e
$n \times m$	\rightarrow	o where $o = n \times m$
$0 \times e$	\rightarrow	0
$1 \times e$	\rightarrow	e
$e \times 0$	\rightarrow	0
$e \times 1$	\rightarrow	e
<code>neg(n)</code>	\rightarrow	$-n$

3 Expression Factory

The definition of expressions in Java becomes rather verbose.

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

It is convenient to introduce short methods that create the expressions. To make them reusable we collect these methods in a factory class.

```
public class ExpressionFactory {
    public static Expression var(String x) {
        return new Variable(x);
    }
}
```

A static import of the factory class lets you use its `public static` members directly. This allows you to write the expression above as follows.

```
import static ExpressionFactory.*;
e1 = add(mul(con(2), con(3)), var("x"));
```

4 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. Your program should then print, evaluate, and optimize the expression. The optimized expression should also be printed and evaluated.

An example of 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+|(?=\\()|(?=\\))|(?=,)|(?<=\\()|(?<=\\))
|(?<=,))";
```

4.1 Submit Your Project

To submit your project, follow these steps.

1. Find the folder that contains your assignment. In Visual Studio Code, you can find this by going to: File → Open Folder. Add the correct folder to a zip file. At the root level, your zip file should contain only one folder, e.g. `assignment-student-start`, which contain the entire project, i.e. the app folder and the Gradle files.
Do not submit only the .java files or the src folder!
2. **Submit this zip file on Brightspace.** Do not submit individual Java files. Only one person in your group has to submit it. Submit your project before the deadline, which can be found on Brightspace.
3. **Do not submit any other format.** Do not submit .rar or .tar.gz or .7z files. Only zip files are allowed.