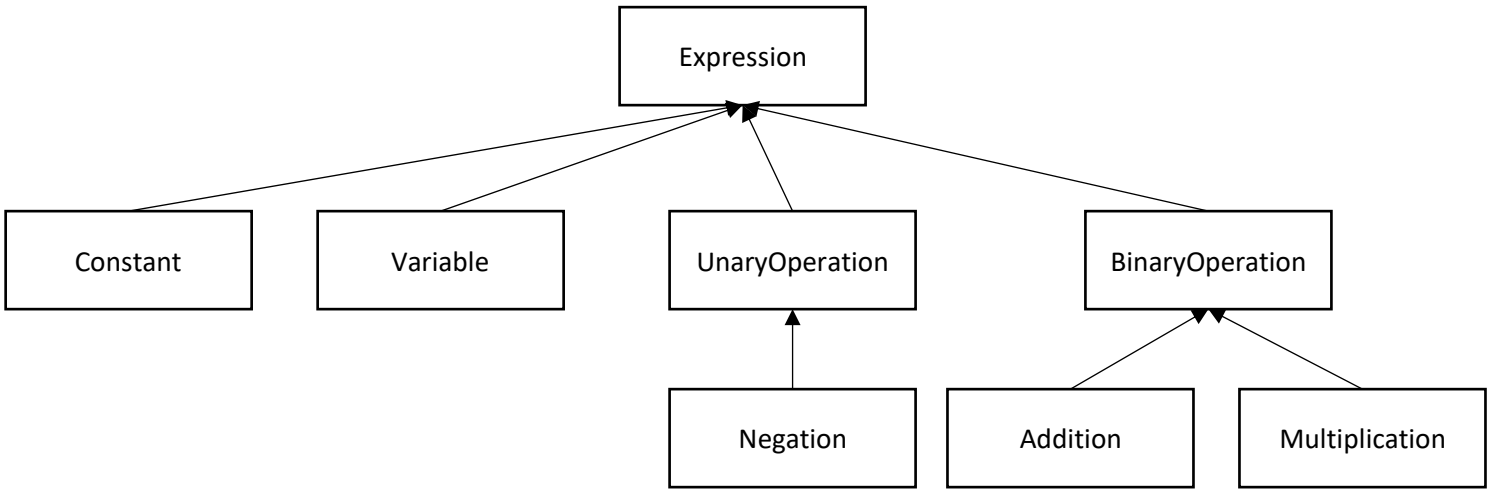
Lab Exercise #3: Expression

In programming, an **expression** is a unit of code that can be evaluated to obtain a value. An expression can contain a constant, a variable, operation, grouping symbol, and function. Below are examples of expressions:

```
x          //Variable expression
12         //Constant expression
(y + 3)    //Addition expression
(z * 3)    //Multiplication expression
-4         //Negated expression
```

In this laboratory, you are to implement expression as a hierarchy of classes as shown in the figure below. We shall only consider expressions that have integer values and limit the operations to negation, addition, and multiplication.



Expression Class

The Expression class is an abstract class that serves as the top-level class of the hierarchy. It contains the following abstract methods:

Method	Description
`abstract Integer getValue()`	Returns the integer value of the expression
`abstract String toString()`	Returns the string representation of the expression

Constant and Variable Classes

The Constant and Variable classes are concrete classes that are direct subclasses of the Expression class. The Constant class represents an expression containing an integer constant. It must implement the following methods:

Method	Description
`Constant(Integer value)`	Initializes a new Constant object with the given integer value

As an example to create a new constant whose value is 3, you can do `Constant c1 = new Constant(3)`.

The class should also override the methods from the Expression class as follows:

Method	Description
`Integer getValue()`	Returns the integer value given in the constructor
`String toString()`	Returns the string representation of the integer given in the constructor

Using the example earlier, the code `c1.getValue()` will return 3 and `c1.toString()` will return "3".

The Variable class is the class for representing an expression that contains only a variable. A variable has a name and can have a value. If no value is explicitly given to a variable, its value is null. It must implement the following methods:

Method	Description
`Variable(String name, Integer value)`	Initializes a new Variable object with the given name and value
`Variable(String name)`	Initializes a new Variable object with the given name. The value of the variable is null
`Integer getName()`	Returns the name of the variable
`void setValue(Integer value)`	Sets the integer value of the variable

The code `Variable v1 = new Variable ("x", 5)` creates a new variable expression with variable `x` initially set to 5. The code `Variable v2 = new Variable("y")` creates a new variable `y` initially set to null. Calling `v1.getName()` returns "x" while performing `v1.setValue(10)` updates the value of variable `x` to 10.

The class should also override the methods of the Expression class as follows:

Method	Description
`Integer getValue()`	Returns the integer value of the variable or null if the variable has not been set before
`String toString()`	Returns the name of the variable

Using the two earlier examples, calling `v1.getValue()` will return 10 while `v2.getValue()` will return null. The code `v1.toString()` will return "x" while `v2.toString()` will return "y".

UnaryOperation and Negation Classes

The UnaryOperation class is an abstract class that extends the Expression class to be able to represent an expression with a unary operator. A unary operator takes only one input expression called the operand. The UnaryOperation class implements the following concrete methods:

Method	Description
`UnaryOperation(Expression operand)`	Initializes a new UnaryOperation object with the given operand
`final Expression getOperand()`	Returns the expression object used as operand in the constructor

The (arithmetic) negation operator is a common unary operator. It takes as input an expression and when evaluated, returns the negated value of the given expression. The Negation class represents a negation expression. It is a concrete subclass of the UnaryOperation class. It does not override any method from the UnaryOperation class, but overrides the methods from the Expression class as follows:

Method	Description
`Integer getValue()`	Returns the negated integer value of the operand or null if the value of the operand is null
`String toString()`	Returns the string representation of the negated version of the operand

As an example, using the expressions from earlier, we can create a new negated expression from them as follows: `Negation n1 = new Negation(c1), n2 = new Negation(v2)`. The code `n1.getOperand()` will return the same object as the one referred to by `c1`. Calling `n1.getValue()` should return -3 since the value of `c1` is 3 while calling `n1.toString()` will return "-3". The result of `n2.getValue()` is null since the value of `v2` is not set (null).

BinaryOperation, Addition, and Subtraction Classes

The BinaryOperation class is an abstract class that represents expressions containing a binary operator. A binary operator is an operator that takes as input two expressions – the left and right operands. This class extends the Expression class by implementing the following concrete methods:

Method	Description
`BinaryOperation(Expression leftOperand, Expression rightOperand)`	Initializes a new BinaryOperation object with the given left and right operands
`final Expression getLeftOperand()`	Returns the expression object used as left operand in the constructor
`final Expression getRightOperand()`	Returns the expression object used as right operand in the constructor

The two common binary operations are addition and multiplication. The Addition class is a subclass of the BinaryOperator that represents a binary expression containing an addition operation. It overrides the methods of the Expression class as follows:

Method	Description
`Integer getValue()`	Returns null if at least one of the two operands is null, otherwise returns the sum of the values from the left and the right operands
`String toString()`	Returns the string representation of the left operand, followed by the addition sign, and then String representation of the right operand, all enclosed in a pair of parentheses

For example, the code `Addition a1 = new Addition(c1, n1)` creates a new addition expression from the two earlier expressions `c1` and `n1`. Calling `a1.getLeftOperand()` will return the same object as `c1`, while `a1.getRightOperand()` will return the same object as `n1`. The value returned by `a1.getValue()` is 0 since the value of `c1` is 3 and `n1` is -3. Calling `a1.toString()` will return the string "(3+-3)".

Let `a1 = new Addition(v1,v2)`. Calling `a1.getValue()` should return null since the value of `v2` is null.

On the other hand, the Multiplication class is a subclass of the BinaryOperator that represents a binary expression containing a multiplication operation. It overrides the methods of the Expression class as follows:

Method	Description
`Integer getValue()`	Returns null if at least one of the two operands is null, otherwise returns the product of the values from the left and the right operands

`String toString()`	Returns the string representation of the left operand, followed by the multiplication sign, and then String representation of the right operand, all enclosed in a pair of parentheses

As an example, the code `Multiplication m1 = new Multiplication (c1, n1)` creates a new multiplication expression from the two earlier expressions `c1` and `n1`. The value returned by `m1.getValue()` is -9 while `a1.toString()` will return "(3*-3)".

Test Client

In this activity you are not going to implement your own client application. Instead, you are to use the included file, ExpressionTester.java. You should not modify the code in this file except for its package declaration. For easier execution of the test client, place the file in the same package as all the Expression classes. Correct implementations of all Expression classes should result to a proper execution of the client with no "Failed" output.

Rubric

Your grade will be based on the following scoring system:

Criteria	Score
Expression Class Hierarchy Implementation	**0pts** for every method that does not perform properly **3pts** for every correctly implemented method but defined in the wrong class **5pts** for every correctly implemented method and defined in the proper class
Client Output	**2pts** for every **Passed** output