# Mini-project 2: BNF expression grammar

***Read the program information; fill in the blanks and draw class diagrams, interaction diagrams (sequence diagrams and/or communication diagrams) for the program.***

**Notes:**

- The below source code is the suggestion, you can freely re-design and re-write the source code. However, in the case of modification, please provide your explanations and reasons for that
- You should enhance the main method with different scenarios so that it can call all necessary methods of other classes
- You may draw several interaction diagrams. Note that the flow of them may illustrate for:
  - the main method and similar scenarios calling other methods
  - complicated methods of some classes (if any)

***Submission: Please send email to [trangntt@soict.hust.edu.vn](mailto:trangntt@soict.hust.edu.vn) before 1 day of the UML3 class.***

- Subject: [Bxx-TSDV-Miniproject] Submission of Miniproject – NguyenVanAn
  - where Bxx is your batch (e.g. B61), NguyenVanAn is your fullname
- Create a folder with your full name and miniproject number, e.g. NguyenVanAn-MP4. Then put the below resources to that folder, compress to a zip file and attach to the email:
  - AnswersForBlanks.txt: Includes your answer for blanks in programs
  - Astah file for class diagrams and interaction diagrams
  - Picture files for class diagrams and interaction diagrams
  - SourceCode folder: export your project to an archive file and put it in this folder.

**[Program Description]**

The following are classes and test class for a simple expression.

(1) The expression grammar is defined in BNF as follows:

```
expression ::= variable | sequence

sequence ::= expression + expression |
             expression - expression  |
             expression * expression |
             expression / expression
```

(2) Method `eval()` is to evaluate the value of an expression. All subclasses of expression need to implement this method.

(3) Method `setValue()` is to set an integer to a variable.

(4) Method `operate()` is to create a new expression by connecting two exist expression with an given operation (e.g. +, -, *, /).

**[Program]**

```
import java.io.*;
interface Expression {
    int eval();
}

class VarExp implements Expression {
    private int var;
    public VarExp() {};
    public void setValue(int n) {
         var = n;

     }

    public int eval() {
                                A
     }

}

class SeqExp implements Expression {
    private int op;
                           B

    public SeqExp(Expression e1, Expression e2, int a_op) {
         exp1 = e1;
         exp2 = e2;
         op = a_op;
     }
    public int eval() {
        switch (op) {
           case 0:
               return exp1.eval() + exp2.eval();
           case 1:
               return exp1.eval() - exp2.eval();
           case 2:
               return exp1.eval() * exp2.eval();
           case 3:
               return exp1.eval() / exp2.eval();
        }
        return 0;
    }
    public SeqExp operate(Expression e, int a_op) {
                         C

    }

}

public class TestExpression {
    public static void main(String args[]) {
        VarExp a = new VarExp();
        VarExp b = new VarExp();
        SeqExp sum = new SeqExp(a, b, 0);
        SeqExp diff = new SeqExp(a, b, 1);
        SeqExp mul = sum.operate(diff,2);
        a.setValue(3);
        b.setValue(7);
        System.out.print(mul.eval());
    }
}
```