1 Submission Instructions

Create a folder named *asuriteid*-p04 where *asuriteid* is your ASURITE user id (for example, if your ASURITE user id is *jsmith6* then your folder would be named jsmith6-p04) and copy all of your *.java* source code files to this folder. Do not copy the *.class* files or any other files. Next, compress the *asuriteid*-p04 folder creating a **zip archive** file named *asuriteid*-p04.zip (e.g., jsmith6-p04.zip). Upload *asuriteid*-p04.zip on the Canvas Module 7: P2 Due submission page before the project deadline. Please see the *Course Summary* section on the Syllabus page in Canvas for the deadline. Consult the Syllabus for the late and academic integrity policies.

2 Learning Objectives

1. Complete all of the learning objects of the previous projects.
2. To implement a GUI interface and respond to action events.
3. To use the linked list, stack, and queue classes.

3 Background

In the lecture notes and video lectures for *Stacks and Queues : Sections 3 – 7* we discussed an application that evaluates an arithmetic expression written in infix notation such as:

```
(-1 - -2) * -(3 / 5)
```

Infix notation is the usual algebraic notation that we are all familiar with where a binary operator (a binary operator is an operator that has two operands) is written between the left-hand and right-hand operands. For example, in the expression above, the left-hand operand of the subtraction operator is -1 and the right-hand operand is -2. Some operators, such as the negation operator, are unary operators meaning there is only one operand (*uni* = one). For negation, the operand is written to the right of the negation operator. Therefore, this expression contains six operators: negation, subtraction, negation, multiplication, negation, and division.

In the algorithm that evaluates the expression, we also treat a left parenthesis as an operator, which it is not, but during the evaluation we have to push left parentheses onto the operator stack.

In an infix arithmetic expression, each operator has a precedence level, which for a left parenthesis, is different when it is on the operator stack as opposed to when it is not (this will become clear when you read the trace of the algorithm for the above expression; see page 3):

Operator	Normal Precedence Level	Stack Precedence Level
(	5	0
–	4	4
* /	3	3
+ –	2	2
)	1	1

Right parentheses really don't have precedence because they are not pushed on the operator stack, but we assign them a precedence level of 1 for consistency. The algorithm discussed in the notes did not handle the negation operator so I have modified it to handle negation. Here is the revised algorithm:

Method *evaluate*(**In:** *pExpr* as an infix expression) **Returns** *Double*
  Create *operatorStack* -- Stores *Operators*
  Create *operandStack* -- Stores *Operands*
  While end of *pExpr* has not been reached **Do**
    Scan next *token* in *pExpr* -- The type of *token* is *Token*

If *token* is an operand **Then**
  Convert *token* to *Operand* object named *number*
  operandStack.push(*number*)
ElseIf *token* is an **InstanceOf** *LeftParen* **Then**
  Convert *token* to *LeftParen* object named *paren*
  operatorStack.push(*paren*)
ElseIf *token* is an **InstanceOf** *RightParen* **Then**
  While not *operatorStack.peek*() is an **InstanceOf** *LeftParen* **Do** *topEval*()
  operatorStack.pop() -- Pops the *LeftParen*
ElseIf *token* is *Negation*, *Addition*, *Subtraction*, *Multiplication*, or *Division* **Then**
  Convert *token* to *Operator* object named *operator*
  While *keepEvaluating*() returns **True Do** *topEval*()
  operatorStack.push(*op*)
End While
While not *operatorStack.isEmpty*() **Do** *topEval*()
Return *operandStack.pop*() -- the result of evaluating the expression
End Method *evaluate*

Method *keepEvaluating*() **Returns True** or **False**
  If *operatorStack.isEmpty*() **Then Return False**
  Else Return *stackPrecedence*(*operatorStack.peek*()) ≥ *precedence*(*operator*)
End Method *keepEvaluating*

Method *topEval*() **Returns Nothing**
  right ← *operandStack.pop*()
  operator ← *operatorStack.pop*()
  If *operator* is *Negation* **Then** *operandStack.push*(-*right*)
  Else *left* ← *operandStack.pop*()
    If *operator* is *Addition* **Then** *operandStack.push*(*left* + *right*)
    ElseIf *operator* is *Subtraction* **Then** *operandStack.push*(*left* - *right*)
    ElseIf *operator* is *Multiplication* **Then** *operandStack.push*(*left* * *right*)
    Else *operandStack.push*(*left* / *right*)
  End If
End Method *topEval*

Method *precedence*(**In:** *Operator pOperator*) **Returns Int**
  If *pOperator* is *LeftParen* **Then Return** 5
  ElseIf *pOperator* is *Negation* **Then Return** 4
  ElseIf *pOperator* is *Multiplication* or *Division* **Then Return** 3
  ElseIf *pOperator* is *Addition* or *Subtraction* **Then Return** 2
  Else Return 1
End Method *precedence*

Method *stackPrecedence*(**In:** *Operator pOperator*) **Returns Int**
  If *pOperator* is *LeftParen* **Then Return** 0
  ElseIf *pOperator* is *Negation* **Then Return** 4
  ElseIf *pOperator* is *Multiplication* or *Division* **Then Return** 3
  ElseIf *pOperator* is *Addition* or *Subtraction* **Then Return** 2
  Else Return 1
End Method *stackPrecedence*

It would be worthwhile to trace the algorithm using the above expression to make sure you understand how it works:

1. Create the operand and operator stacks. Both are empty at the beginning.
2. Scan the first token (and push it onto the operator stack.
3. Scan the next token – (negation) and push it onto the operator stack.
4. Scan the next token 1 and push it onto the operand stack.
5. Scan the next token – (subtraction). Since the operator on top of the operator stack (negation) has higher precedence than subtraction, evaluate the top (note: negation is a unary operator so there is only one operand to be popped from the operand stack):
   a. Pop the top number from the operand stack. Call this *right* = 1.
   b. Pop the top operator from the operator stack. Call this *operator* = – (negation).
   c. Evaluate *operator* and push the result (-1) onto the operand stack.
   d. Now push the subtraction operator onto the operator stack.
6. Scan the next token – (negation). Since the operator on top of the stack (subtraction) has precedence less than negation, push the negation operator onto the operator stack.
7. Scan the next token 2 and push it onto the operand stack.
8. Scan the next token). Pop and evaluate operators from the operator stack until the matching (is reached.
   a. The top operator is a unary operator (negation):
      Pop the top number from the operand stack. Call this *right* = 2.
      Pop the top operator from the operator stack. Call this *operator* = – (negation).
      Evaluate *operator* and push the result (-2) onto the operand stack.
   b. The top operator is a binary operator (subtraction):
      Pop the top number from the operand stack. Call this *right* = -2.
      Pop the top number from the operand stack. Call this *left* = -1.
      Pop the top operator from the operator stack. Call this *operator* = – (subtraction).
      Evaluate *operator* and push the result (1) onto the operand stack.
   c. The top operator is (so pop it.
9. Scan the next token * (multiplication). The operator stack is empty so push *.
10. Scan the next token – (negation). Since negation has higher precedence than the operator on top of the operator stack (multiplication) push the negation operator onto the operator stack.
11. Scan the next token (and push it onto the operator stack.
12. Scan the next token 3 and push it onto the operand stack.
13. Scan the next token / (division). Since the operator on top of the stack (left parenthesis) has higher precedence than division push / onto the operator stack. Now do you see why the precedence of (changes when it is on the operator stack?
14. Scan the next token 5 and push it onto the operand stack.
15. Scan the next token). Pop and evaluate operators from the operator stack until the matching (is reached.
   a. The top operator is binary operator (division):
      Pop the top number from the operand stack. Call this *right* = 5.
      Pop the top number from the operand stack. Call this *left* = 3.
      Pop the top operator from the operator stack. Call this *operator* = /.
      Evaluate *operator* and push the result (0.6) onto the operand stack.
   b. The top operator is (so pop it.
16. The end of the infix expression string has been reached. Pop and evaluate operators from the operator stack until the operator stack is empty.
   a. The top operator is a unary operator (negation):
      Pop the top number from the operand stack. Call this *right* = 0.6.
      Pop the top operator from the operator stack. Call this *operator* = – (negation).
      Evaluate *operator* and push the result (-0.6) onto the operand stack.

b. The top operator is a binary operator (multiplication):
   Pop the top number from the operand stack. Call this *right* = -0.6.
   Pop the top number from the operand stack. Call this *left* = 1.
   Pop the top operator from the operator stack. Call this *operator* = *.
   Evaluate *operator* and push the result (-0.6) onto the operand stack.

17. The operator stack is empty. Pop the result from the operand stack (-0.6) and return it.

4 Software Requirements

The project shall implement a GUI calculator which accepts as input a syntactically correct arithmetic expression written in infix notation and displays the result of evaluating the expression. The program shall meet these requirements.

1. The program shall implement a GUI which permits the user to interact with the calculator. Watch the Project 4 video lecture for a demonstration of how the application works.
2. When the Clear button is clicked, the input text field and the result label shall be configured to display nothing.
3. When a syntactically correct infix arithmetic expression is entered in the input text field and the Evaluate button is clicked, the program shall evaluate the expression and display the result in the label of the GUI.
4. When the input text field is empty, clicking the Evaluate button does nothing.
5. When the Exit button is clicked, the application shall terminate.
6. **Note:** you do not have to be concerned with syntactically incorrect infix expressions. We will not test your program with such expressions.

5 Software Design

Refer to the UML class diagram in Section 5.21. Your program shall implement this design.

5.1 Main Class

The *Main* class shall contain the *main*() method which shall instantiate an object of the *Main* class and call *run*() on that object. *Main* is completed for you.

5.2 AddOperator

Implements the addition operator, which is a binary operator. *AddOperator* is completed for you. Use *AddOperator* as a guide when completing *DivOperator*, *MultOperator*, and *SubOperator*.

5.3 BinaryOperator

The abstract superclass of all binary operators. *BinaryOperator* is completed for you. Note that *BinaryOperator* implements one abstract method *evaluate*() which all subclasses must implement. The subclasses are *AddOperator*, *DivOperator*, *MultOperator*, and *SubOperator*.

5.4 DivOperator

Implements the division operator, which is a binary operator. Complete the code in this file by using the *AddOperator* class as an example.

5.5 DList<E>

This is the *DList<E>* class from the *Module 7 Source Code* zip archive. It implements a **generic** doubly linked list where the data type of each list element is *E*. *DList<E>* is completed for you. For example, to create a *DList* which stores elements of the type *Token* you would write `DList<Token> list = new DList<>();` much in the same way that we can create an *ArrayList* of *Double*s by writing `ArrayList<Double> list = new ArrayList<>();`

5.6 Expression

Represents an infix expression to be evaluated. Use the provided pseudocode as a guide in completing this class.

5.7 LeftParen

Represents a left parenthesis in the expression. *LeftParen* is completed for you. Note that *LeftParen* is a subclass of the abstract class *Parenthesis*.

5.8 MultOperator

Implements the multiplication operator, which is a binary operator. Complete the code in this file by using the *AddOperator* class as an example.

5.9 NegOperator

Implements the negation operator, which is a unary operator. Complete the code in this file by using the *AddOperator* class as an example. Note, however, that negation is a unary operator so it only has one operand.

5.10 Operand

An operand is a numeric value represented as a *Double*. Implement the class using the UML class diagram as a guide.

5.11 Operator

Operator is the abstract superclass of all binary and unary operators, i.e., it is the superclass of *BinaryOperator* and *UnaryOperator*. Implement the class using the UML class diagram as a guide. Note that all of the non-constructor methods are abstract, i.e., none of them are implemented in *Operator*.

5.12 Parenthesis

Parenthesis is the superclass of *LeftParen* and *RightParen*. These are treated as a weird sort of *Operator* because we need to be able to push *LeftParen*s on the operator stack when evaluating the expression. *Parenthesis* is completed for you.

5.13 Queue<E>

Implements a generic queue data structure using a *DList<E>* list to store the elements. This is the same class that was provided in the *Week 7 Source* zip archive. *Queue* is completed for you.

5.14 RightParen

Represents a right parenthesis in the expression. *RightParen* is completed for you.

5.15 Stack<E>

Implements a generic stack data structure using a *DList<E>* list to store the elements. This is the same class that was provided in the *Module 7 Source* zip archive. *Stack* is completed for you.

5.16 SubOperator

Implements the subtraction operator, which is a binary operator. Complete the code in this file by using the *AddOperator* class as an example.

5.17 Token

Token is the abstract superclass of the different types of tokens (i.e., symbols) that can appear in an infix expression. *Token* is completed for you.

5.18 Tokenizer

The *Tokenizer* class scans a *String* containing an infix expression and breaks it into tokens. For this project, a token will be either an *Operand* (a double value), a *LeftParen* or *RightParen*, or an arithmetic *UnaryOperator* (subclass *NegOperator*) or *BinaryOperator* (one of the *AddOperator, SubOperator, MultOperator*, or *DivOperator* subclasses). *Tokenizer* is completed for you. It is implemented as a **finite state machine** (FSM) which are commonly used in computing, especially when breaking a "sentence" of words or symbols into its component parts. If you have the time, you should study the code to learn how FSM's work and are used.

5.19 UnaryOperator

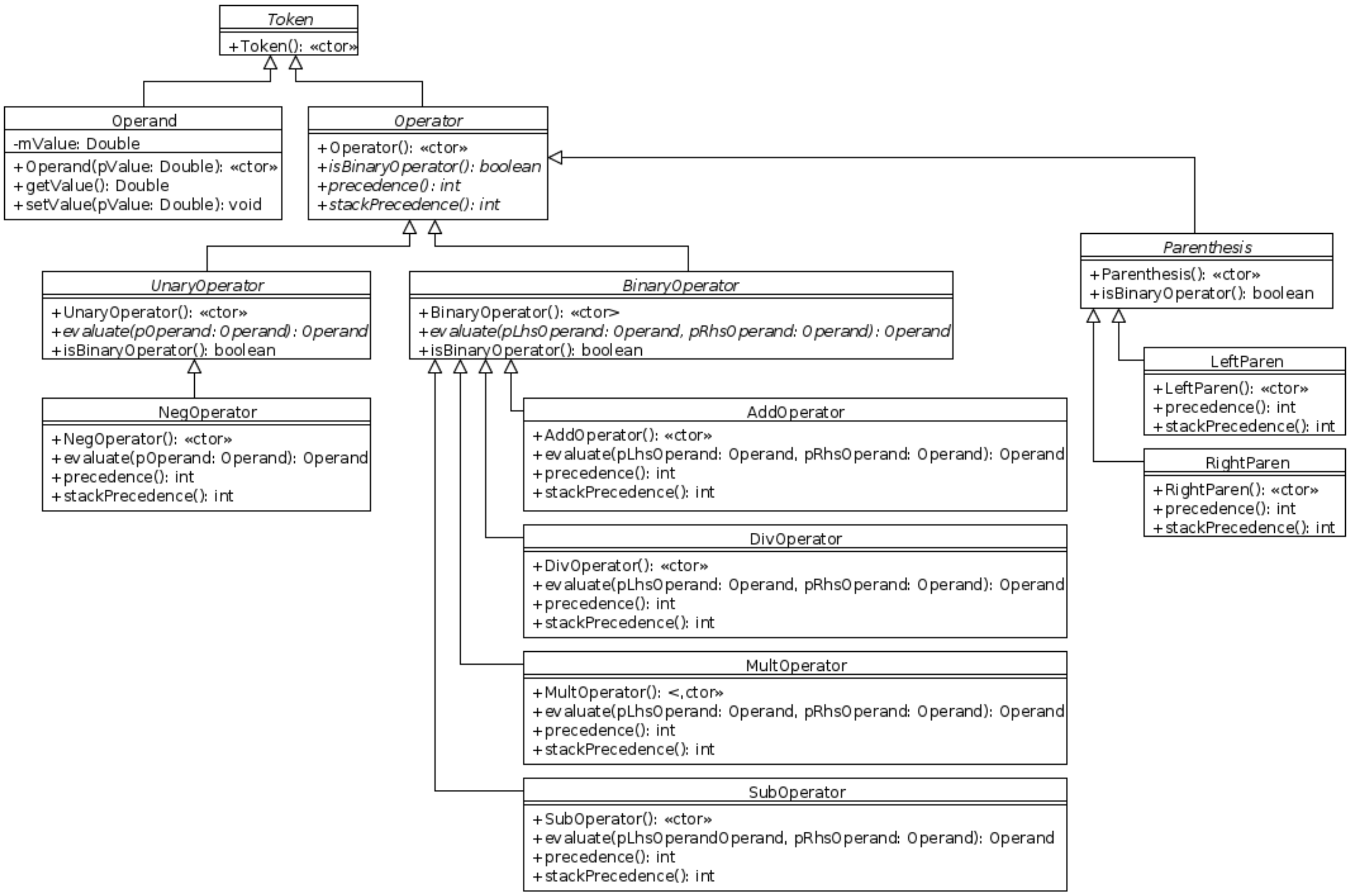
UnaryOperator is the superclass of all unary operators. *UnaryOperator* is completed for you.

5.20 View

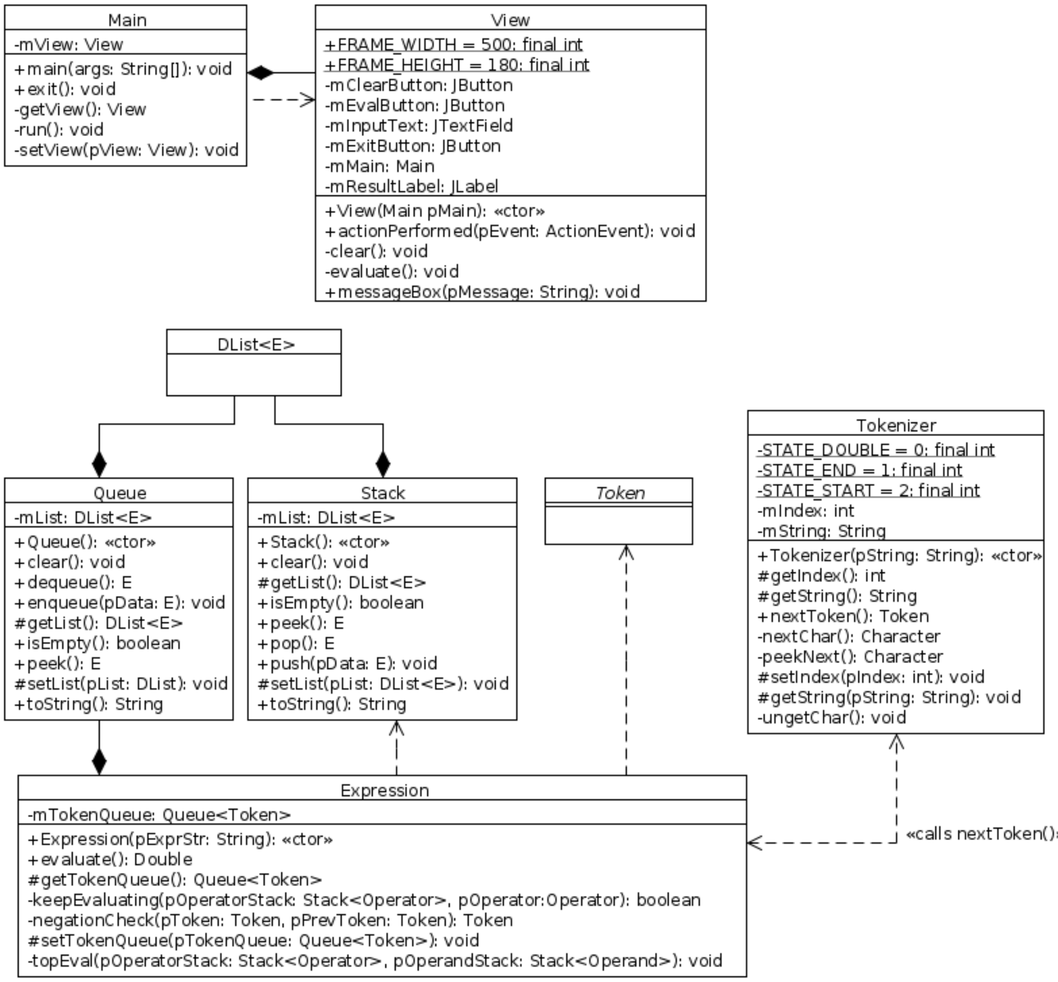
The *View* implements the GUI. Read the comments and implement the pseudocode.

5.21 UML Class Diagram

The UML class diagram is provided in the zip archive */uml* folder as two UMLet files. Because the images in this document are small, there are PNG images of the two class diagrams in the */image* folder. Your program shall implement this design.



5.21 UML Class Diagram (continued)



6 Additional Project Requirements

1. Format your code neatly. Use proper indentation and spacing. Study the examples in the book and the examples the instructor presents in the lectures and posts on the course website.
2. Put a comment header block at the top of each method formatted thusly:

```
/**
 * A brief description of what the method does.
 */
```

3. Put a comment header block at the top of each source code file formatted thusly (or use /** ... */ comments if you wish):

```
//**************************************************************************************************************
// CLASS: classname (classname.java)
//
// COURSE AND PROJECT INFO
// CSE205 Object Oriented Programming and Data Structures, semester and year
// Project Number: project-number
//
// AUTHOR: your-name, your-asurite-id, your-email-addr
//**************************************************************************************************************
```