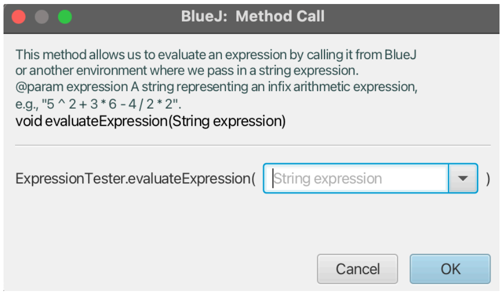
User Manual

Operating Systems

By: Jacob Attia

How to run the code:

- Run the ExpressionTester Class
    - Right click the class, it will prompt the “void evaluateExpression(String expression)”
    - You then will be prompted with this screen



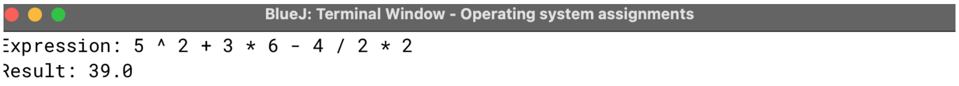
This method allows us to evaluate an expression by calling it from BlueJ or another environment where we pass in a string expression.
@param expression A string representing an infix arithmetic expression, e.g., "5 ^ 2 + 3 * 6 - 4 / 2 * 2".
void evaluateExpression(String expression)

ExpressionTester.evaluateExpression(String expression)

- You then want to fill in any expression by starting it and ending it with quotation marks as well as leaving a space between each number and expression
- For example: “5 ^ 2 + 3 * 6 - 4 / 2 * 2”

Results:

- After running the code correctly you should be prompted with this screen



```
Expression: 5 ^ 2 + 3 * 6 - 4 / 2 * 2
Result: 39.0
```

- The console will consist of the following:
    - The expression you put in for the program to solve
    - The results of the expression you put

Steps in code creation:

1. The data structure
    - Each internal node can be an operator
    - Each leaf node is a number
2. Tokenize the expression
    - Split the string expression (“5 ^ 2 + 3 * 6”) in tokens
    - [“5”, “^”, “2”, “+”, “3”, “*”, “6”]
    - Use expression.split to handle space separated tokens
3. Operator Precedence
    - In java, ^ has the highest precedence, then * and /, then + and -
    - Assign each operator a numerical precedence
    - ^ → 3, * | / → 2, +/- → 1
    - We do this because it is easier to compare precedence when we build the tree
    - If the current operator is equal or lower in the precedence then the operator at the top of the stack, we pop and form a subtree first
4. Stacks to build the tree
    - Stack for nodes hold partial subtrees or single number nodes
    - If we see “5”, we push a node with value “5”
    - Stack for operators hold symbols like +, -, *, /, ^
5. Process of the stacks
    - Read a token
    - If its a number, make a node and push it to the node stack
    - If its an operator, compare it with the top of the operator stack
    - If the top has equal or higher precedence, pop it and build a subtree using the top two nodes in the node stack
    - Push the new subtree back onto the node stack
    - Then push the current operator onto the operator stack
6. Node class
    - We need a blueprint for each tree element
    - String value; → to store either a number or an operator

   - Node left, right; → references to the left and right child nodes for a binary tree
   - Two constructors: one for lead nodes and on for operator nodes
7. Evaluating the tree
   - If the nodes value is a number, just convert it to double and return it
   - If the nodes value is an operator:
   - Evaluate the left child → leftVal
   - Evaluate the right child → rightVal
   - Apply the operator to leftVal and rightVal
8. ExpressionTester class
   - Keep the tree logic in ExpressionTree but create a separate class to:
   - Accept a string expression
   - Split into tokens
   - Call the tree-building method
   - Call the evaluation method
   - Print the result
   - Main Method:
   - Minimal usage, just prints instructions or can be customized to handle hardcoded expressions or user input
   - In BlueJ, we rely on calling evaluateExpression interactively with a string
9. Final check
   - Test code by compiling then running which will then lead to filling out an expression in the correct format