README

This program is a symbolic calculator that can compute arithmetic expressions.

Usage

To run:

>> java Calculator

Then input a math expression, math equation, or one of the three keywords (help, history, exit).

- If you input a math expression, the program will simplify it and display the answer.
- If you input a math equation, the program will tell you if it is true or not
- If you input “help”, the program will show a tutorial
- If you input “history”, the program will show a log of what you’ve done already
- If you input “exit”, the program will exit.

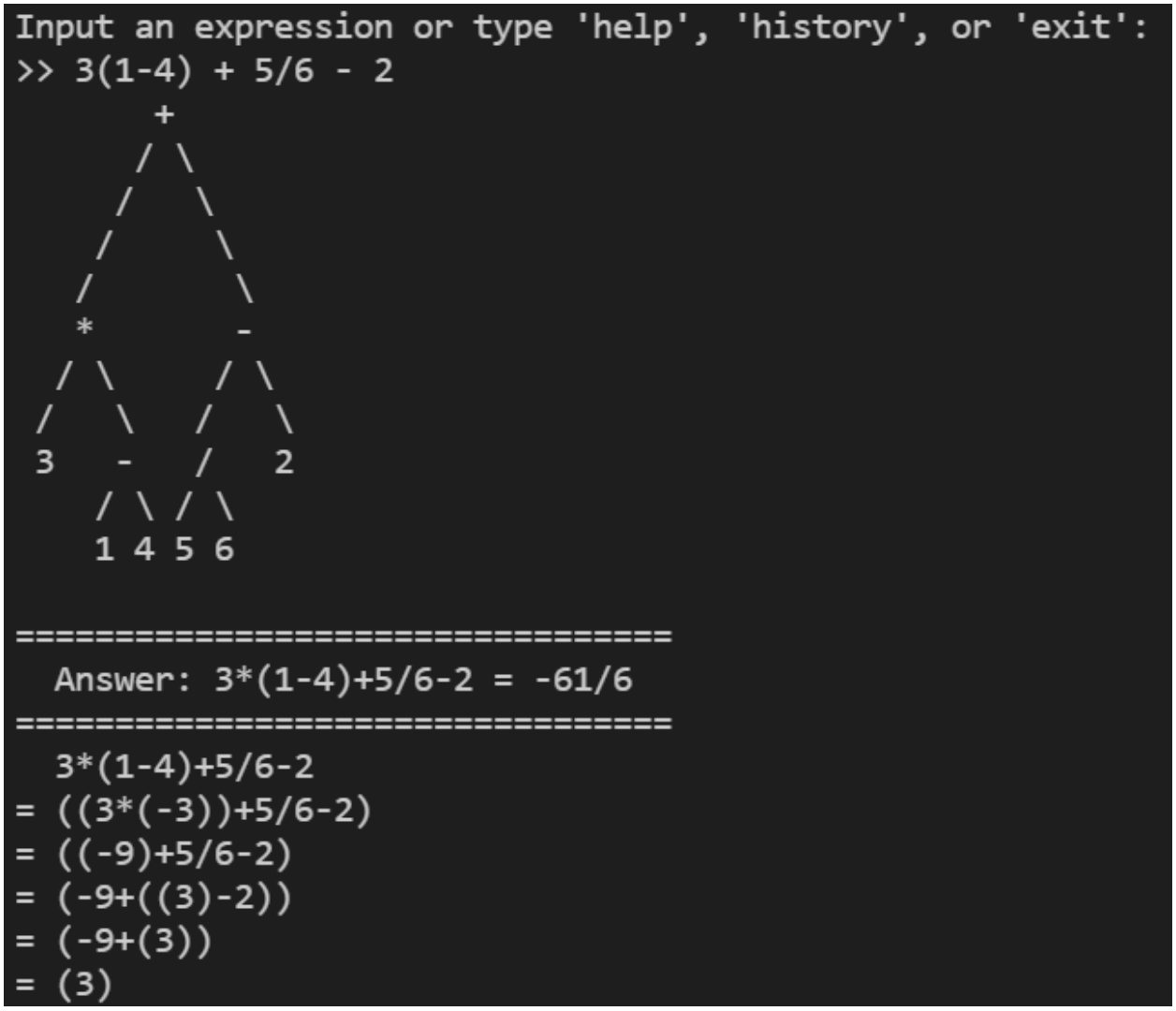
In the expressions, you can use integers and standard arithmetic operators: + - * /
You can also use parentheses.

For the equations, input two expressions separated by a “=” in the middle.

Some other usage notes:

- The program only works with integers, not decimals.
- A lot of the complexity in this program comes from keeping fractions as fractions. If you add, multiply, subtract, or divide fractions, the program will symbolically keep track of the numerators and denominators, and will give you an answer that is a fraction with the lowest possible denominator.
- The program also shows you the steps it took to get to the final answer. The final answer is correct, but sometimes the in-between steps are missing necessary parentheses.

An example run:



Correct and efficient use of inheritance with at least one superclass and two subclasses

The Node class is the superclass. The Node class is used to make a binary tree which represents a math expression. It defines a value for the node and references to the left and right children. It defines a printTree(), which nicely prints the tree, and a isLeaf() function, which detects whether or not the node instance has children.

```
public class Node {

    public Node left;
    public Node right;

    public int value;
    ...
```

```
    /**
     * Displays an ASCII-art sort of representation of this tree
(image-like text).
     */
    public void printTree() {
        Node root = this; //for the purposes of printing it
        int maxLevel = maxLevel(root);
        List<Node> rootList = new ArrayList<Node>();
        rootList.add(root);
        printNodeInternal(rootList, 1, maxLevel);
    }
```

```
    /* If is leaf in the tree */
    public boolean isLeaf() {
        return (left == null && right == null);
    }
```

There are also several stub functions which are overridden by subclasses. The most significant one is simplify().

```
    /** The following functions will be overridden by subclasses */

    /**
    * Creates a copy of itself which is a simplified version of itself and
its children
     */
    public Node simplify() {
        return null;
    }
```

```
    /* Display this node as text */
    public String text() {
        return "";
    }

    /* Get the value of this node */
    public int value() {
        return value;
    }

    /* If is operator (to be further defined in subclasses) */
    public boolean isOperator() {
        return false;
    }

    /* If is variable (to be further defined in subclasses) */
    public boolean isVariable() {
        return false;
    }

    /* If is a parentheses (to be further defined in subclasses) */
    public boolean isParenthesis() {
        return false;
    }
```

The Node class is extended by 4 different subclasses. I'll focus on Number and Operator.

The Number class overrides the stub functions with the correct values for a number:

```
    public boolean isOperator() {
        return false;
    }
    public boolean isVariable() {
        return false;
    }
```

It also defines simplify() as returning a copy of the Number with the same value. This is specific to numbers, because we don't want other types of nodes in the tree to simplify the same way.

```
    public Node simplify() {
        return new Number(value);
    }
```

The Operator class, on the other hand, goes through a 100-line process to **simplify()** itself and its children. (lines 37-166)

This process involves:

- Checking whether both, neither, or only one of its children are Numbers
- Applying the specific operation (for examples, subtraction) to its children
- If its children is the division operator, creating one or more fractions, applying its own operator, and then simplifying the result.

Correct and efficient use of two other class concepts

The main class concept used here is the binary tree. I've already shown how the Node has its own value and references to its children.

```
Input an expression or type 'help', 'history', or 'exit':
>> 12 + 13 + 14 * 15
   +
  / \
 /   \
 +   *
/ \ / \
12 13 14 15

============================
  Answer: 12+13+14*15 = 235
============================
  12+13+14*15
= ((25)+14*15)
= (25+(210))
= (235)
```

The program also uses a HashMap to record the log of what the user has entered, and what the answers were.

In action:

```
Input an expression or type 'help', 'history', or 'exit':
>> history
12+13+14*15 --> 235
2-1 --> 1
6 --> 6
3*(1-4)+5/6-2 --> -61/6
```

The history is a HashMap because the order of the elements doesn't matter. What matters is that a particular input mapped to a particular output, and a HashMap is a good way of showing that.

You can find the HashMap implemented in Calculator.java

```
HashMap<String, String> history = new HashMap<>();
```

```
 if(simple.hasErrorMessage) {
                    System.out.println(simple.errorMessage);
                } else {
                    String result = "Answer: "+root.text() +" =
"+simple.text();
                    history.put(root.text(), simple.text());
                    System.out.println(boxMessage(result));
                }
```

Sufficiently substantive project

To get this project to output simple numbers, this project makes use of a lot of recursion. The simplify() process essentially starts at the bottom of the tree (the leaves) and simplifies as much as possible before moving up the tree one node, and simplifying as much as possible then. This happens all the way to the root node, so that by the end, the root nodes' children are either just one number, or two numbers divided by each other.

A similar recursion process happens for the Node's text() method.

Good style and organization, including JavaDocs style methods comments

The code is organized into classes based on its functionality, for example Parser has its own class to convert math Strings into a binary tree. Variable names are informative, and methods have JavaDocs style comments.

```
    /**
     * This method takens in math text and parses it into a tree of Nodes
     * @param input the String, which should only contain numbers,
arithmetic,
                    possibly an 'x', and whitespace
        @return root of tree. Each Node will be of the correct subclass
depending on what
                they parse out to be.
     */
    public Node parse(String input) {
```

```
    /** The main functionality of this subclass. Searches children to
simplify it and them as
     * much as possible. If children are Numbers, then this function will
simply compute the
     * number answer. If children are other Operators, then recursively
calls them to simplify()
      @return the simplified Node
    */
    public Node simplify() {
```