CSC 413.02

Assignment 1: Calculator with GUI

Alonzo Contreras:
https://github.com/CSC-413-SFSU-02/csc413_02_p1-alonzocontreras

Introduction

In this assignment I was tasked of taking a skeleton code (pre made code that one needs to add and fix in order for it to operate accordingly) provided to me and implement a working calculator with a GUI (Graphical User Interface). A GUI is an interface that allows humans to interact with the program. I will be creating an outline of how I will approach the task at hand while using a flow chart to visualize the process. For the IDE (Integrated Development Environment: a software application that gives the tools to the programmer to develop and execute code) I used NetBeans 8.1 to develop and execute the program. Using NetBeans Swing (Swing is a GUI widget toolkit for Java) I was able to finish developing and executing the calculator GUI.

In order to compile and execute the program, one must download all the files from the repository and open them on NetBeans as a New Java Project using Existing Sources. Once in NetBeans, in order to execute, just press the Play button on top toolbar.

Assumptions

Before I started writing the code, I started by reading the code and tried to understand what the skeleton code was doing, or trying to do. A good way to understand what a certain code is doing is by tracing the outline backwards to see how the hierarchy is built. From what I read, I assumed the skeleton code had a working While Loop that used a tokenizer function that scanned the string of operators and operands and placed them accordingly into their respected stack which I assumed were functional. Therefore I was able to conclude that I needed to create classes for each operator and code their functions for each those operators. I then created a HashMap that kept each operator and their priority depending on what operator it is. For example, ^ has the highest priority, then * /, then + -, then ( ).

Implementation

To begin my implementation I started with the Operand Class where I filled in each method in the skeleton code to where the Operand would properly enter the stack. In the check method I decided to check if the token matched one of the operators instead of checking for a number since there are so many numbers to choose from but only 8 operators. This would return false but I inverted the 'If Statement' in the eval class to be true.

```
public class Operand {

     int value;

    public Operand(String token) {
            this.value = Integer.valueOf(token);
    }

    public Operand(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }

    public static boolean check(String token) {
        return token.matches("\\+|\\-|\\*|\\/|\\#|\\(|\\)|\\^");
    }
}
```

```
if (!Operand.check(token)) {
    operandStack.push(new Operand(token));
```

After doing so, I moved over to the Operator Class to implement a HashMap that would store keys and their values; in this case the operators themselves and their priority level.

```
public abstract class Operator {

    public int priority;

    private static HashMap <String, Operator> operators = new HashMap();

    static {
        operators.put("+", new AdditionOperator());
        operators.put("-", new SubtractionOperator());
        operators.put("/", new DivisionOperator());
        operators.put("*", new MultiplicationOperator());
        operators.put("^", new ExponentOperation());
        operators.put("(", new OpenParenthesisOperator());
        operators.put(")", new ClosingParenthesisOperator());
        operators.put("#", new PoundOperator());
    }

    public abstract int priority();

    public abstract Operand execute(Operand op1, Operand op2);

    public static boolean check(String token) {
        return operators.containsKey(token);
    }

    public static Operator whichOperator(String token){
        return operators.get(token);
    }

}
```

Once I created the HashMap, I created a Class for every operator in the HashMap in order to set the priority level as well as the actual function for each operator. For example in the Addition Operator, the Class extends the Operator class, meaning that it is the child class of Operator.

```
class AdditionOperator extends Operator {

    public AdditionOperator() {
        priority = 2;
    }

    @Override
    public int priority() {
        return priority;
    }

    @Override
    public Operand execute(Operand op1, Operand op2) {
      return new Operand(op1.value + op2.value);
    }

}
```

Once both Operand and Operator Classes were completed I moved over to the Evaluator Class and worked on the eval function that performed the main task, which was checking and calling the correct functions. I first pushed the pound sign "#" into the Operator Stack to use it as a check to see if I ran out of operators to compute. This allowed me to avoid running into errors like EmptyStackException. I then modified the Operator initializer since the original line was broken due to the fact that one could not initialize from an abstract class in which Operators is. The original line was

```
Operator newOperator = new Operator();
```

and I rewrote it to:

```
Operator newOperator = Operator.whichOperator(token);
```

where I created a method in operator that allows me to check which operator is being called and this allows me to initialize from an abstract class. (Seen at the bottom of the Operator Class)

```
public abstract class Operator {

    public int priority;

    private static HashMap <String, Operator> operators = new HashMap();

    static {
        operators.put("+", new AdditionOperator());
        operators.put("-", new SubtractionOperator());
        operators.put("/", new DivisionOperator());
        operators.put("*", new MultiplicationOperator());
        operators.put("^", new ExponentOperation());
        operators.put("(", new OpenParenthesisOperator());
        operators.put(")", new ClosingParenthesisOperator());
        operators.put("#", new PoundOperator());
    }

    public abstract int priority();

    public abstract Operand execute(Operand op1, Operand op2);

    public static boolean check(String token) {
        return operators.containsKey(token);
    }

    public static Operator whichOperator(String token){
        return operators.get(token);
    }

}
```

I then created two ‘If Statements’ in order to check for both the Open Parenthesis and the Closing Parenthesis. The Open Parenthesis check allowed the operator to be immediately pushed into the Operator Stack and the Closing Parenthesis check allowed me to call a new method that I created to evaluate all the operands and operators between the two parenthesis.

```
if (token.equals("(")) {
    operatorStack.push(newOperator);
    continue;
}
if (token.equals(")")) {
    this.evaluateParenthis();
    continue;
}
```

```
private void evaluateParenthis() {
    while (!(operatorStack.peek().priority == 1)) {
        Operator oldOpr = operatorStack.pop();
        Operand op2 = operandStack.pop();
        Operand op1 = operandStack.pop();
        operandStack.push(oldOpr.execute(op1, op2));
    }
    operatorStack.pop();
}
```

Once I was able to evaluate the equation between the parenthesis, I needed to modify the While Loop given in the skeleton code that evaluated two operands and an operator. I wrapped the While Loop by an ‘If Statement’ and inverted it in order to check to see the operator that’s on top of the Operator Stack is the pound sign “#”. I checked it by looking for the priority of the operator being checked, in where I initially made the pound sign priority to 0.

```
if (!(operatorStack.peek().priority == 0)) {
    while (operatorStack.peek().priority() > newOperator.priority()) {
        Operator oldOpr = operatorStack.pop();
        Operand op2 = operandStack.pop();
        Operand op1 = operandStack.pop();
        operandStack.push(oldOpr.execute(op1, op2));
    }
}
operatorStack.push(newOperator);
```

In the case that the operator that was being checked was not the pound sign, then it would jump in the while loop and checked if the operator in the stack’s priority was higher than the operator that’s being compared/ready to be pushed into the stack. If it is greater, i.e. the * is on top of the stack and the

operator being compared was the +, then it will evaluate the two numbers by * and then push the result back into the Operand Stack. But in the chance the operator in the stack had a lower priority then it would immediately push the operator that was being compared to into the stack.

Once all the tokens in the equation have been checked and ran through the main While Loop, I used a second While Loop that computes the remaining operands and operators that were not evaluated in the first While Loop discussed above. In this While Loop, I was checking again if the operator on top of the stack's priority is 0, meaning if it's the pound sign "#", then it will know it has no more operators to use to evaluate. Once it evaluates the remainder, it will then pop the pound sign out of the stack and then return the value of the operand left in the Operand Stack which is the final result.

```
        while (!(operatorStack.peek().priority == 0)) {
            Operator oldOpr = operatorStack.pop();
            Operand op2 = operandStack.pop();
            Operand op1 = operandStack.pop();
            operandStack.push(oldOpr.execute(op1, op2));
        }
        operatorStack.pop();
        return operandStack.pop().getValue();
    }
}
```

Conclusion

After working with this assignment, I have come to realize that my ability to read other people's code is not as strong as my ability to write new code. What I struggled with the most is fixing the line:

Operator newOperator = new Operator();

I didn't understand abstract classes as much I thought I did, and I spent a majority of time researching and experimenting on how to get it to work. Only to come to realize, that I needed to create a separate method that was abstract in order to initialize from an abstract class. What I also learned and still practicing is planning out a project instead of diving straight into coding. This allows me to plan out the classes and methods needed in order to make the project run and saves me a lot of time from doing trial and error. In addition, this helped me brainstorm ideas that I wouldn't originally would of come up with. Overall I struggled completing this project, but it is the first step into becoming a better programmer.