**Approach, design & algorithm**

Before starting this project I carefully looked over all of the requirements. The goals for two of the methods were, generally, the same as the algorithms discussed in the textbook. Therefore, for the conversion from infix to postfix and the evaluation of postfix expressions I rewrote my understanding of what the book had covered. I tried my best to write the algorithm from memory and only refer to the project material for additional guidance. After implementing each of these algorithms I ensured that the algorithms threw exceptions as the JUnit tests expected.

However, after passing all public JUnit tests for these methods I implemented additional safeguards. For example, passing “(1 + 2) 1” to my infix to postfix method did not throw any exception. Yet, this format is clearly invalid as it results in “12+1” which will cause an error upon evaluation. This expression has n operands and n-2 operators, however, all algebraic expressions should have n operands and n-1 operators, assuming multiplication is always explicitly stated. Similarly, my infix to postfix method would treat “2++++++2” as valid. This previous input has a very similar issue to the other input discussed, It does not have n operands and n - 1 operator. Thus, to resolve both of these issues it is best to add a counter for operators and operands to evaluate the validity of the expression after evaluating each character in the infix string. Another issue I found is if you pass “1 + 2 ( \* 3” the postfix expression will output “123\*(+” which clearly should not be legal. To safeguard against this, after iterating through the entire infix string I check to see if any of the items is an open parenthesis. If so, I throw an exception because that signifies that the expression had more open parentheses than close parentheses. On the other hand, I was unable to find any severe failure cases, that were not tested by the JUnit tests, for the postfix evaluation method.

At the time of planning, I was unaware that the documentation for this project provided an algorithm for converting from postfix to infix. Thus, I utilized my problem-solving skills to try and piece together some sort of methodology to complete this task. My first inclination was that this algorithm would be similar to the evaluation method because we needed to do the same thing up until the actual calculation of “operand1 operator operand2.” Therefore, my algorithm began as pushing to the stack whenever an operand was encountered. From there, I took a look at the JUnit test cases to reverse engineer exactly how the test cases wanted the solution to look. After this, I ran my initial algorithm against some simple and intermediate test cases by hand. Of course, my algorithm did not work exactly as expected so I tweaked it and went back to testing. I did this a couple of times until I ultimately landed on an algorithm that worked for a few intermediate cases. After this, I ran it against the JUnit test for further validation. Later, I was pleasantly surprised to find out that my algorithm worked exactly like the one provided. Ultimately, this was an unnecessary exercise in program design and problem-solving because the algorithm was already available in the documentation, however, this brain teaser was a very welcome addition to this project.

**What did you learn?**

Before this project, I had no experience with postfix expressions much less conversion to and from this expression type. Therefore, I learned a lot about postfix expressions that I did not know about before this project. It is interesting to see how many more unpredictable/unusual situations come up when you are processing infix expressions when compared to the number of unusual situations that can occur when processing postfix expressions.

The subject of this project, infix and postfix expressions is all new to me. However, the concepts we utilized to achieve our goals such as static methods, switch cases, while loops, etc are core concepts I know and understand well. I do not say that to devalue the benefit I derived from this subject matter but rather to provide a better idea of what portion of this project provided the most learning opportunities. Nonetheless, the application of these core concepts is extremely useful for reinforcing my understanding of them.

**What issues did you encounter, if any?**

As alluded to in the design section of this write up I did struggle a little bit with the postfix to infix algorithm given that I did not see the algorithm was provided for us and thus brainstormed and wrote it myself. Although, this was not a major issue for me.

Another issue that briefly stumbled me was how to compare the precedence of operators. Initially, I wanted to see if I could apply some of the logic I am learning in CMSC 207 to take a complicated statement form comparing two operators and convert it into a much simpler statement form. That is, I had planned to create a long string of conditionals to compare two operators and try to reduce it using the laws of logical equivalence such as De Morgan’s laws. However, I soon realized that this process would be much easier if I could represent operators with the same precedence as a singular entity. In other words, “+” or “-” would be treated as the same as they have the same precedence. I quickly realized that to do this I would need a switch statement that would convert both operators into their “reduced” form and then use conditionals to compare them. After trying to work through some of the logic I began realizing that having a method that both converts from a character operator to a precedence level and then compares two different operators would really have two separate “functionalities.” Thus, my next thought was to separate these functionalities into distinct methods. Ultimately, I came to the realization that I could simply define a method that would convert the precedence of a character operator into an integer and then use comparison operators on the client side, a solution that was far removed from my initial logic reduction approach.

**What would you have done differently?**

I am not completely satisfied with how I throw exceptions in my infix to postfix method. At this moment I throw exceptions within the conversion method as I convert to enforce the validity of the initial infix string. This would not be an issue if there were simply one or two invalidating conditions, but it looks like there are a multitude of ways that an infix expression could be invalid, which would cause my method to behave unexpectedly. Thus, if I would have had time I would have liked to create a private **validateInfix** method that would check the validity of an infix string before passing it onto the conversion step. Although this validation would require a similar loop structure and switch statement as the conversion method I believe it is more logical to separate these functionalities.

On the other hand, the validation step would have to loop through the whole string once meaning it would have a time efficiency of O(n) while the conversion step would need to perform a similar loop of all the characters thus granting it the same efficiency of O(n). Therefore, validation and conversion would become an O(2n) operation. Of course, for very large n O(n) and O(2n) would behave relatively the same but it is still important to think about the minor efficiency loss that would be traded in for more logical and readable code.

**How can you apply this concept in the future?**

Quite honestly, It is hard to come up with any pressing reason to work with postfix expressions outside of how a computer actually processes algebraic expressions. I understand that it is much easier for a computer to process postfix expressions compared to infix expressions. As alluded to earlier, this reality is hinted at by the number of cases that can cause an exception when processing infix expressions when compared to processing postfix expressions. However, I do not think I will be in a position where I will be coding how a computer treats algebraic expressions anytime soon.

Nonetheless, It would be interesting to know where postfix expressions can be useful outside of improving a computer’s efficiency.

**Anything else you would like to share?**

I have nothing else to share at this moment.

**Questions**

I have no pressing questions to ask at this moment.

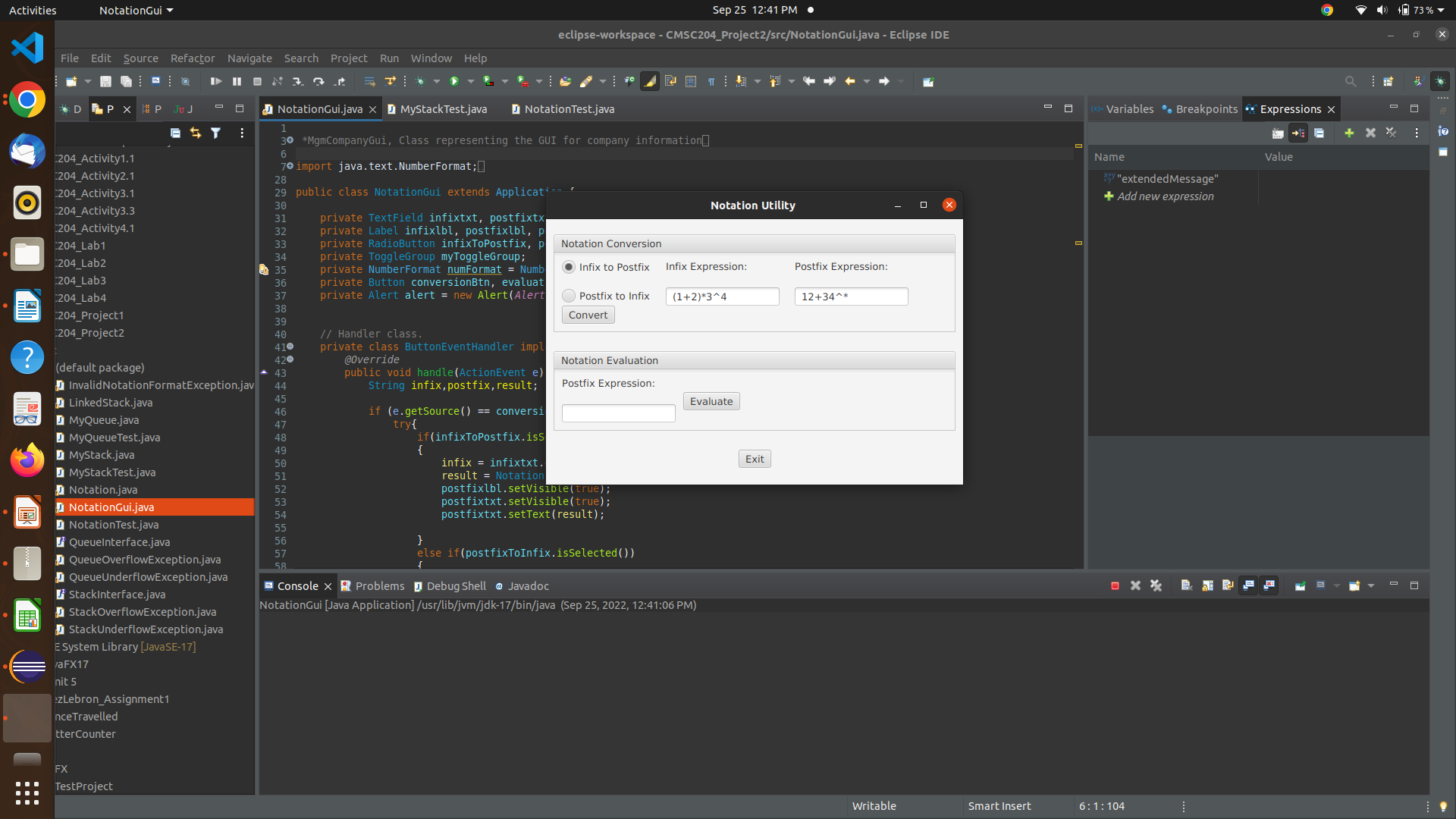
**Assumptions**

I am assuming that this class will be used **exclusively** to process one-digit postfix and infix expressions. I would have loved to pursue the two-digit implementation but because of time constraints I was unable to.

**Test cases (infix to postfix method):**

case: “(1+2)\*3^4”

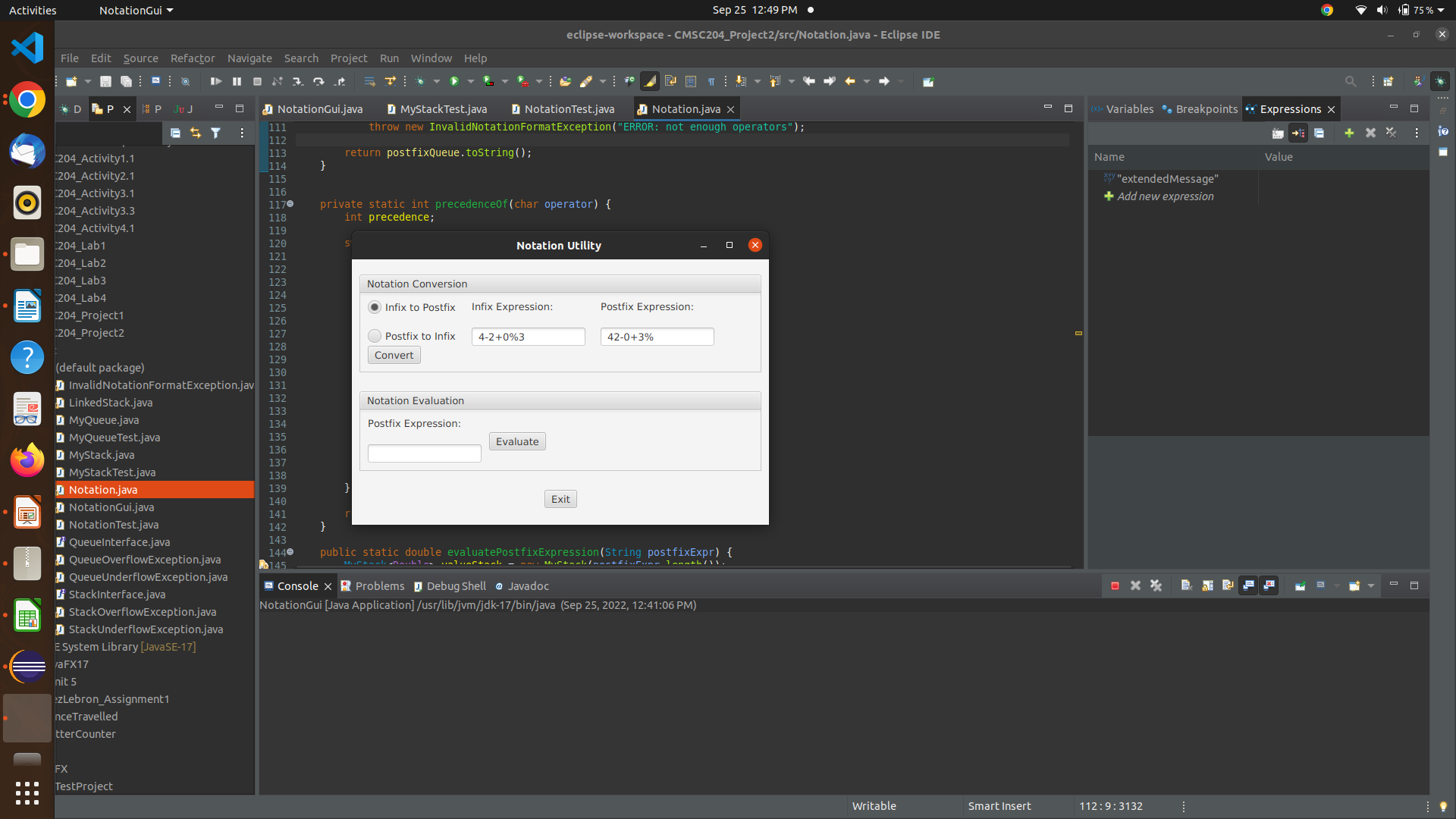
expected: “12+34^\*”

result: “12+34^\*”

case: “4-2+0%3”

expected: “42-0+3%”

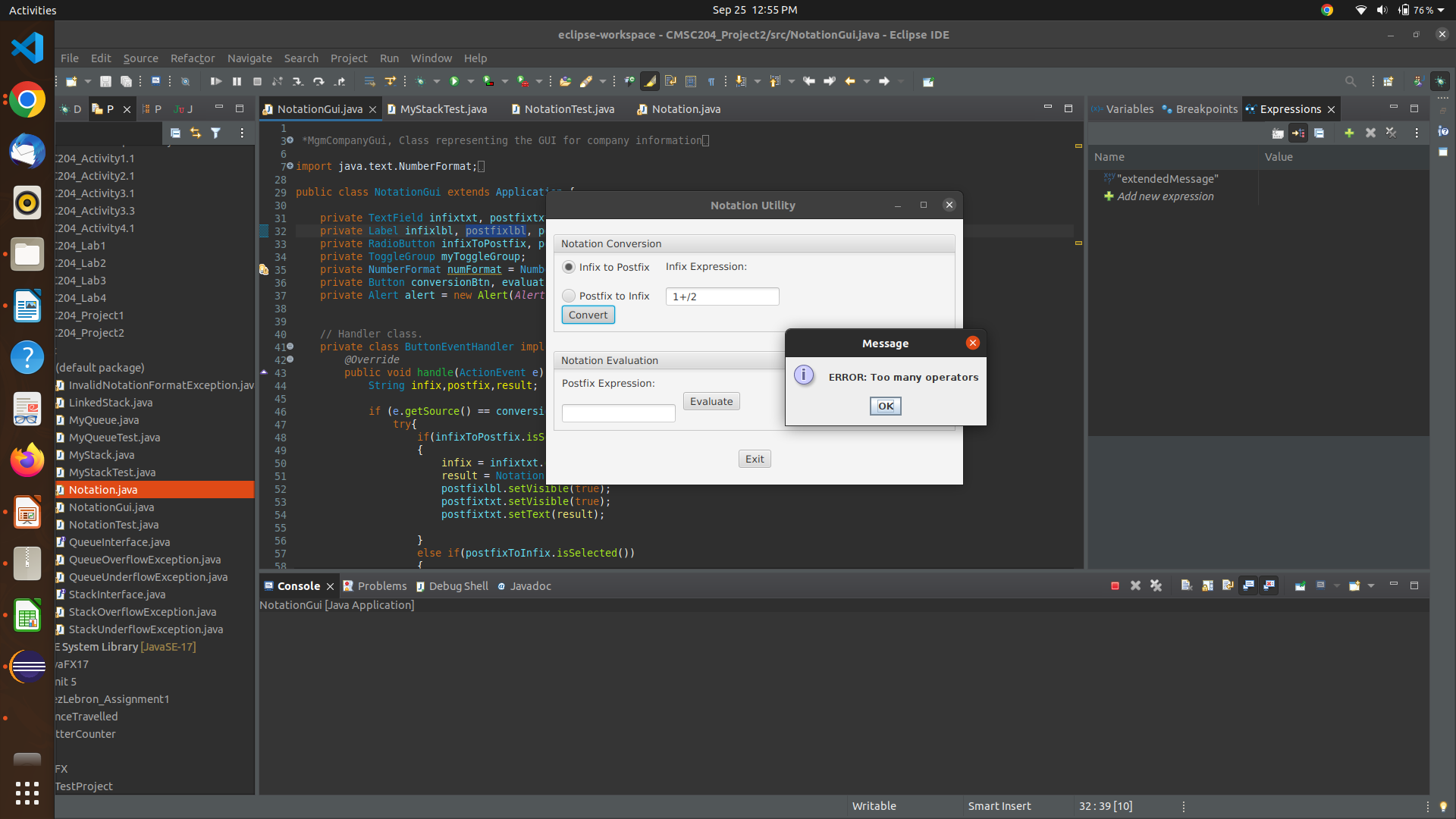
result: “42-0+3%”



case: “1+/2”

expected: InvalidNotationFormatException

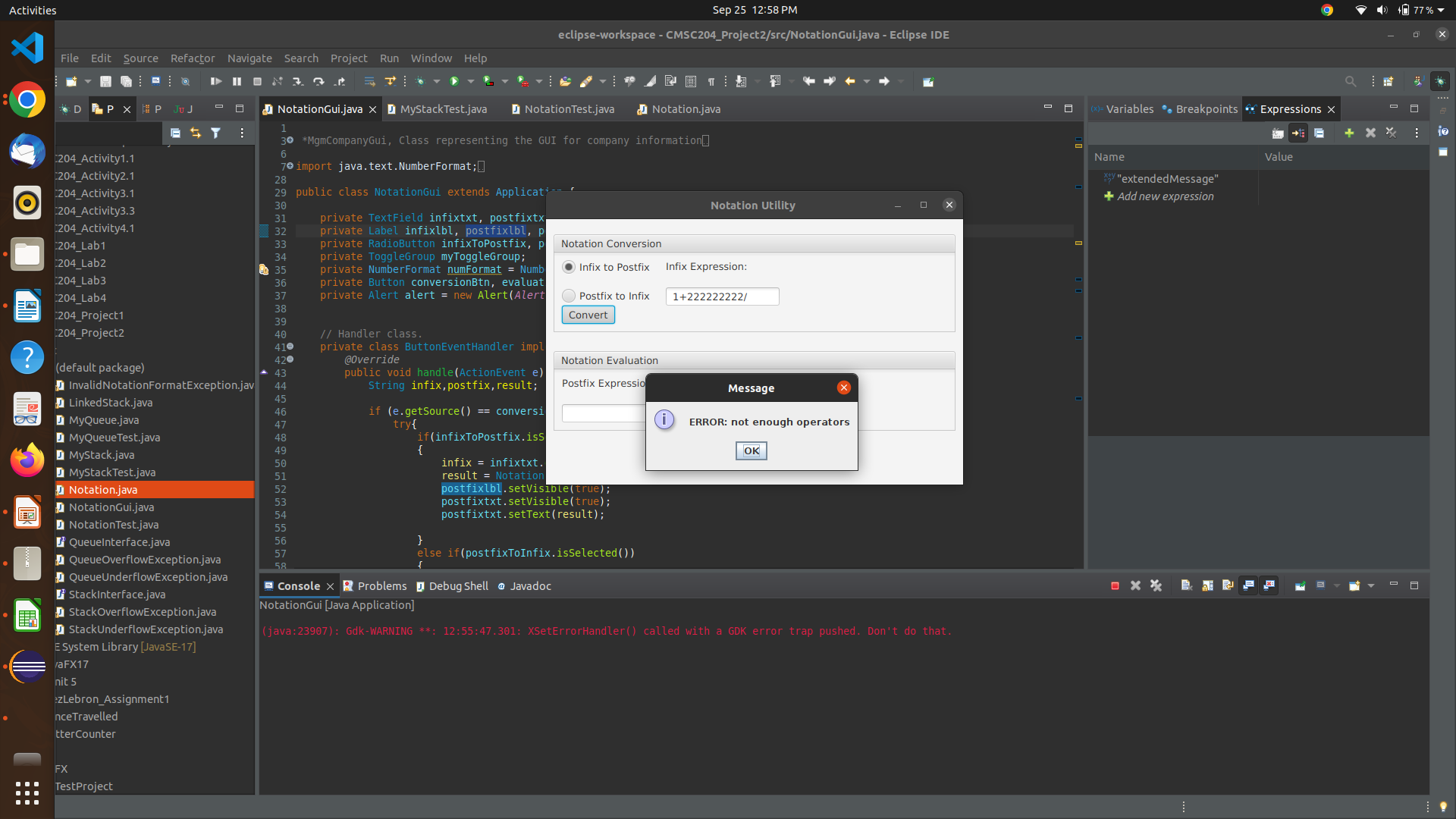
result: InvalidNotationFormatException



case: “1+222222222/”

expected: InvalidNotationFormatException

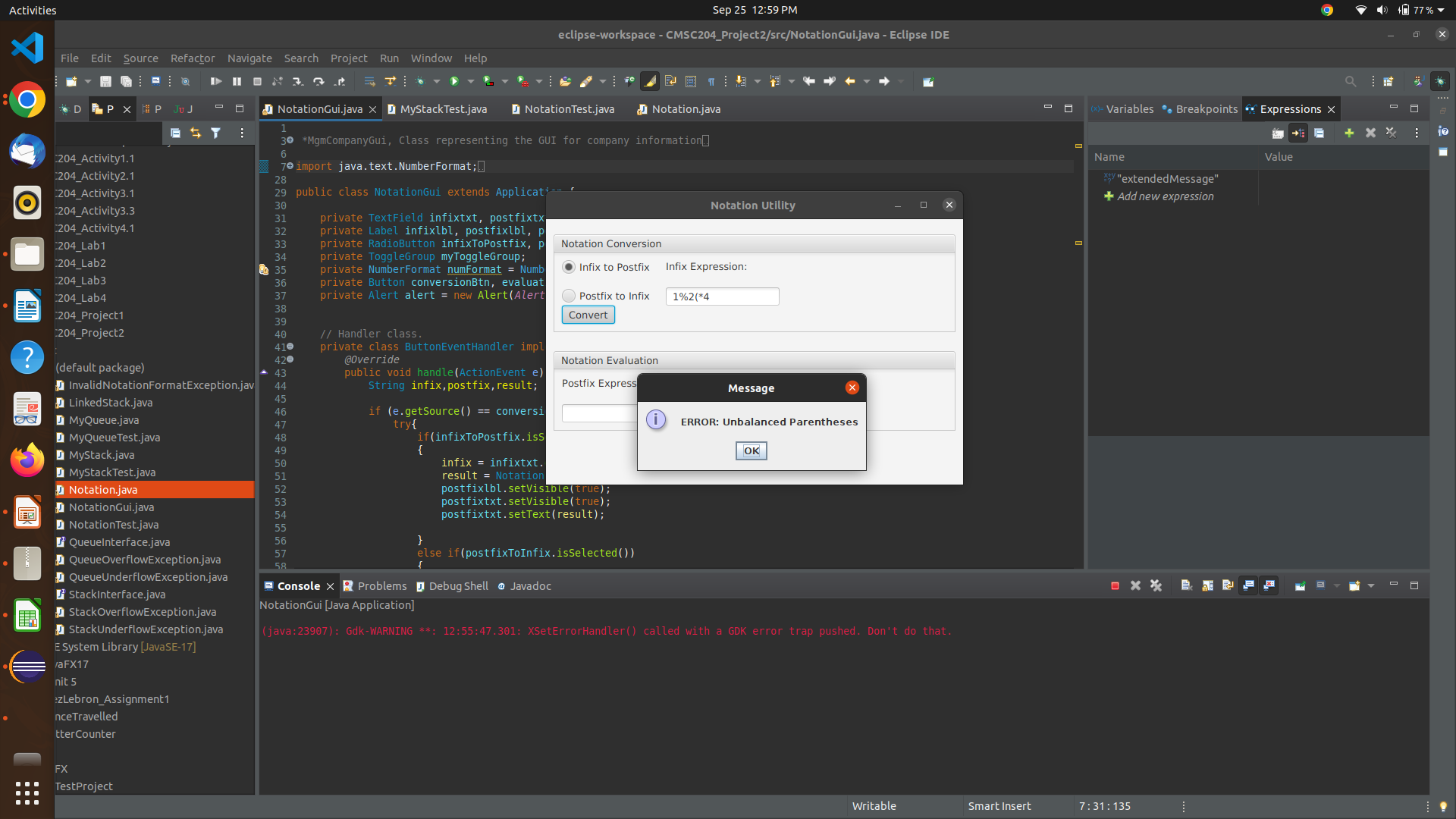
result: InvalidNotationFormatException



case: “1%2(\*4”

expected: InvalidNotationFormatException

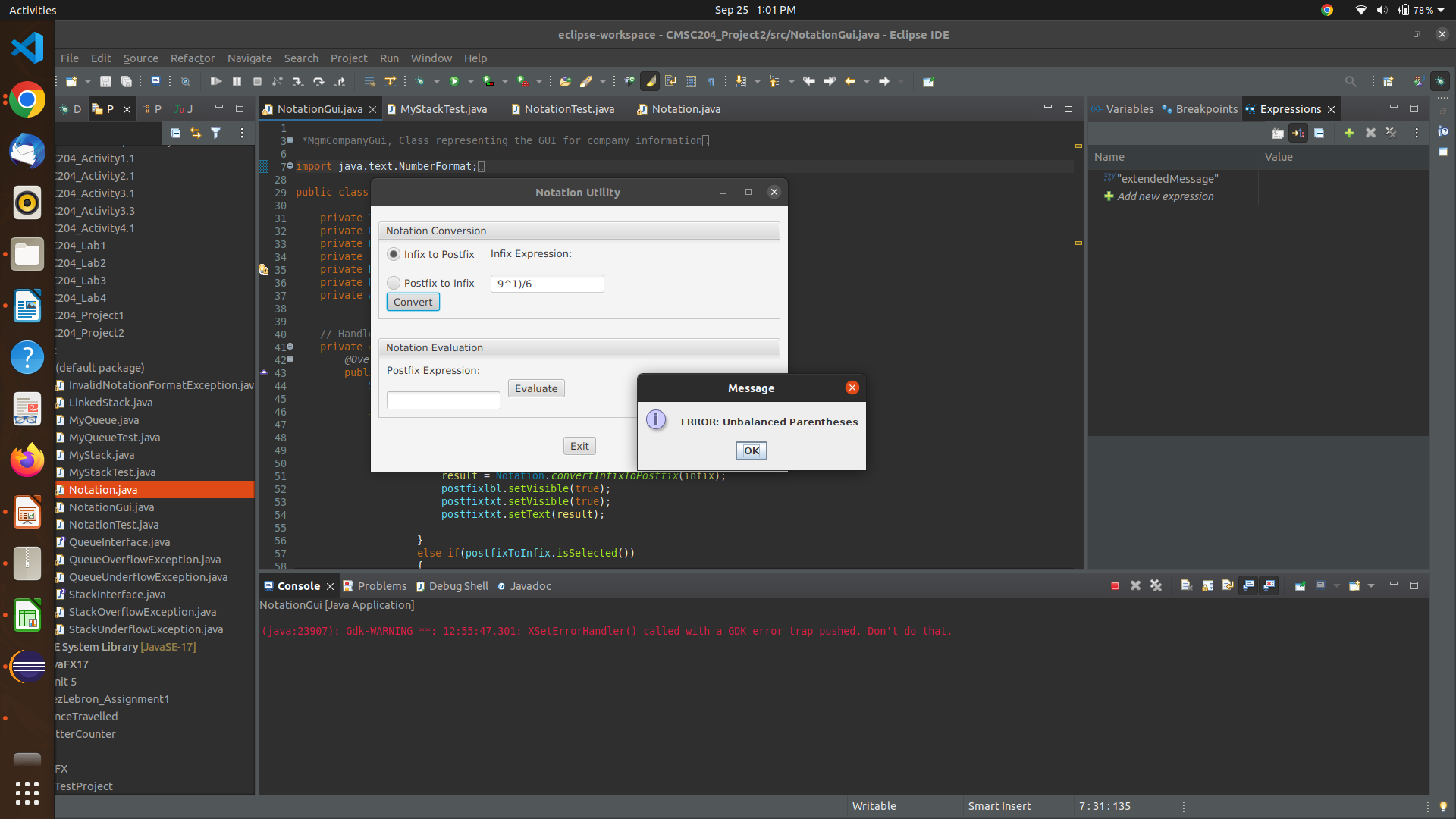
result: InvalidNotationFormatException



case: “9^1)/6”

expected: InvalidNotationFormatException

result: InvalidNotationFormatException

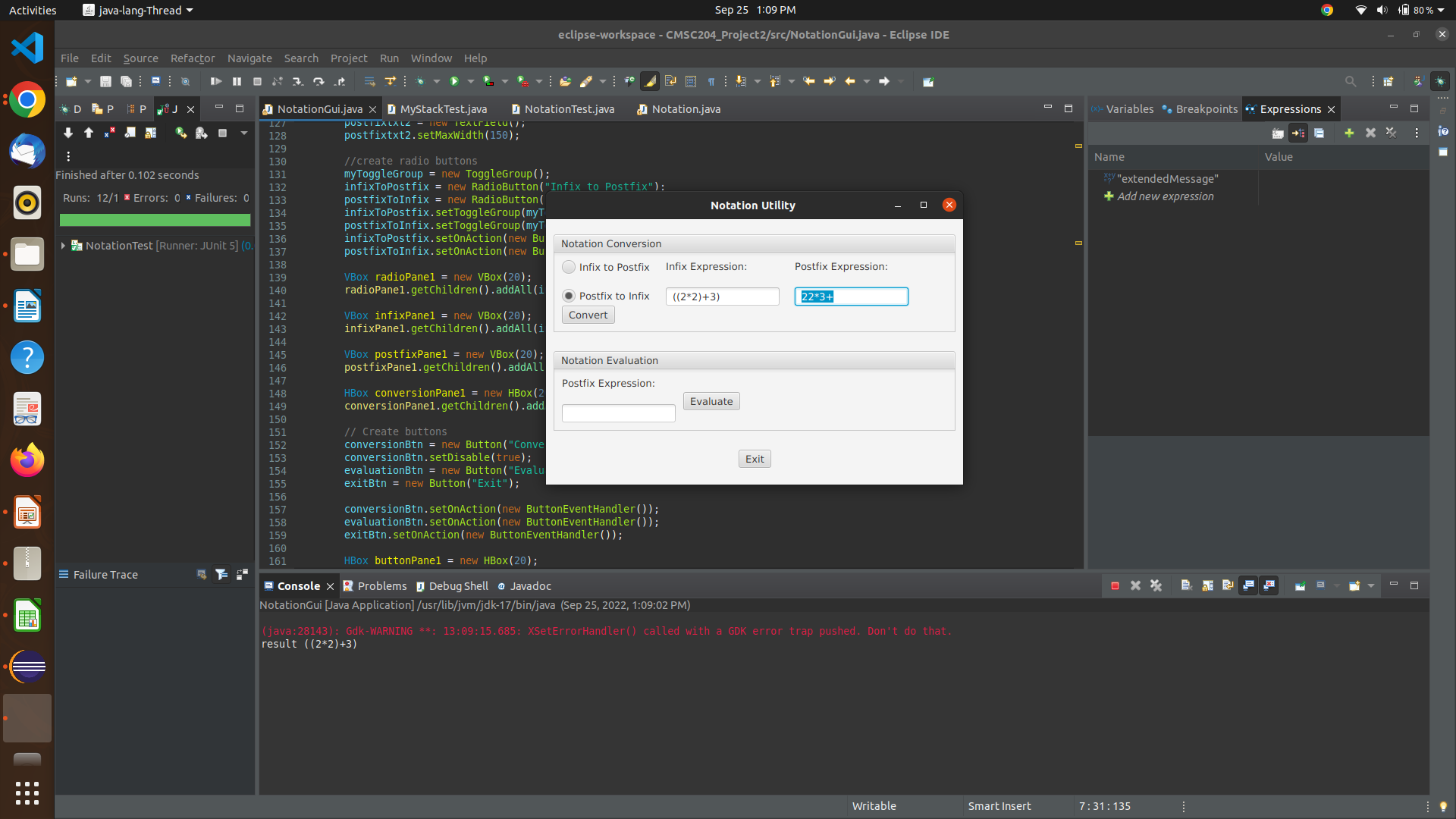


**Test cases (postfix to infix method):**

case: “22\*3+”

expected: “((2\*2)+3)"

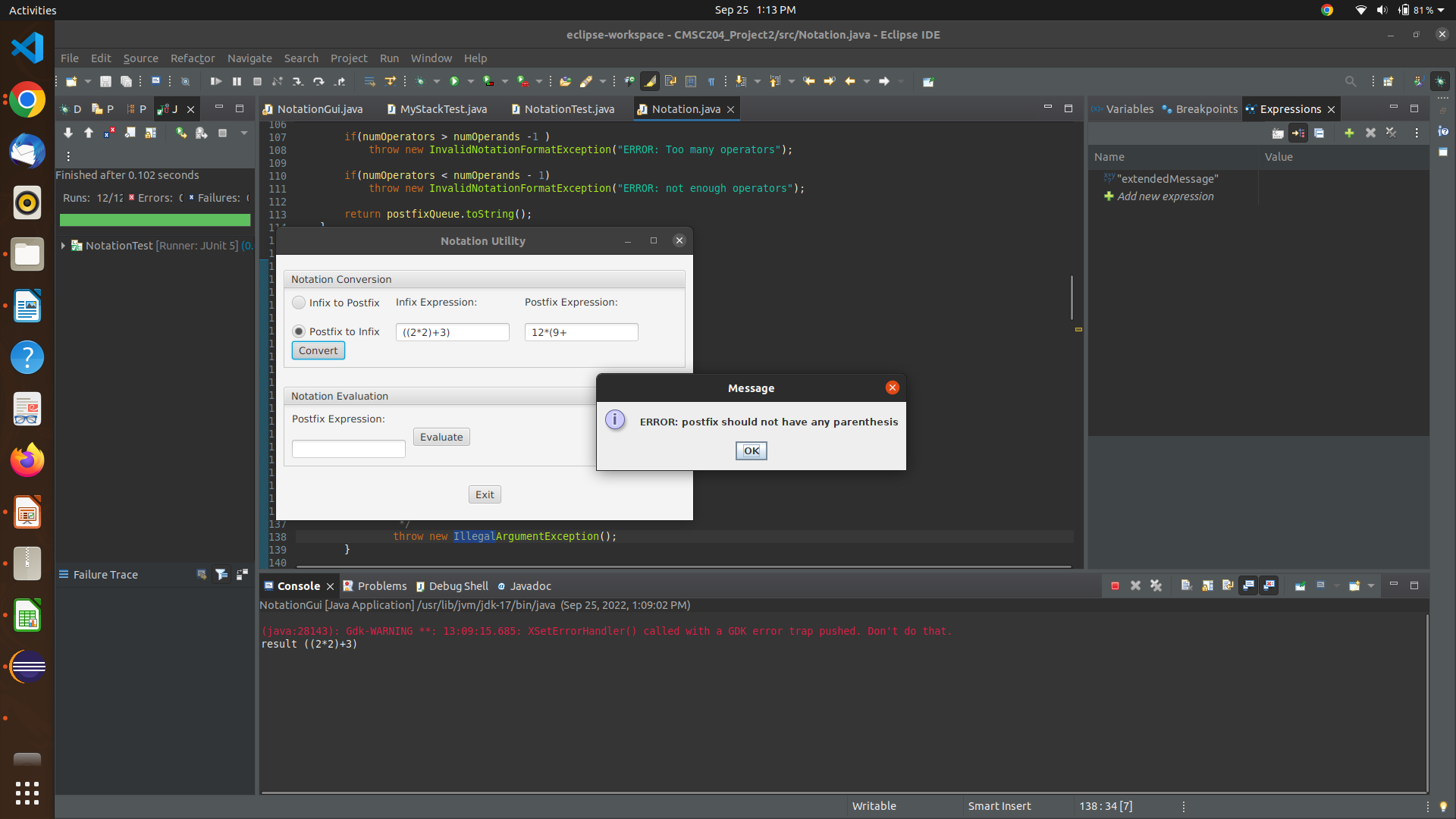
result: “((2\*2)+3)”



case: “12\*(9+”

expected: InvalidNotationFormatException

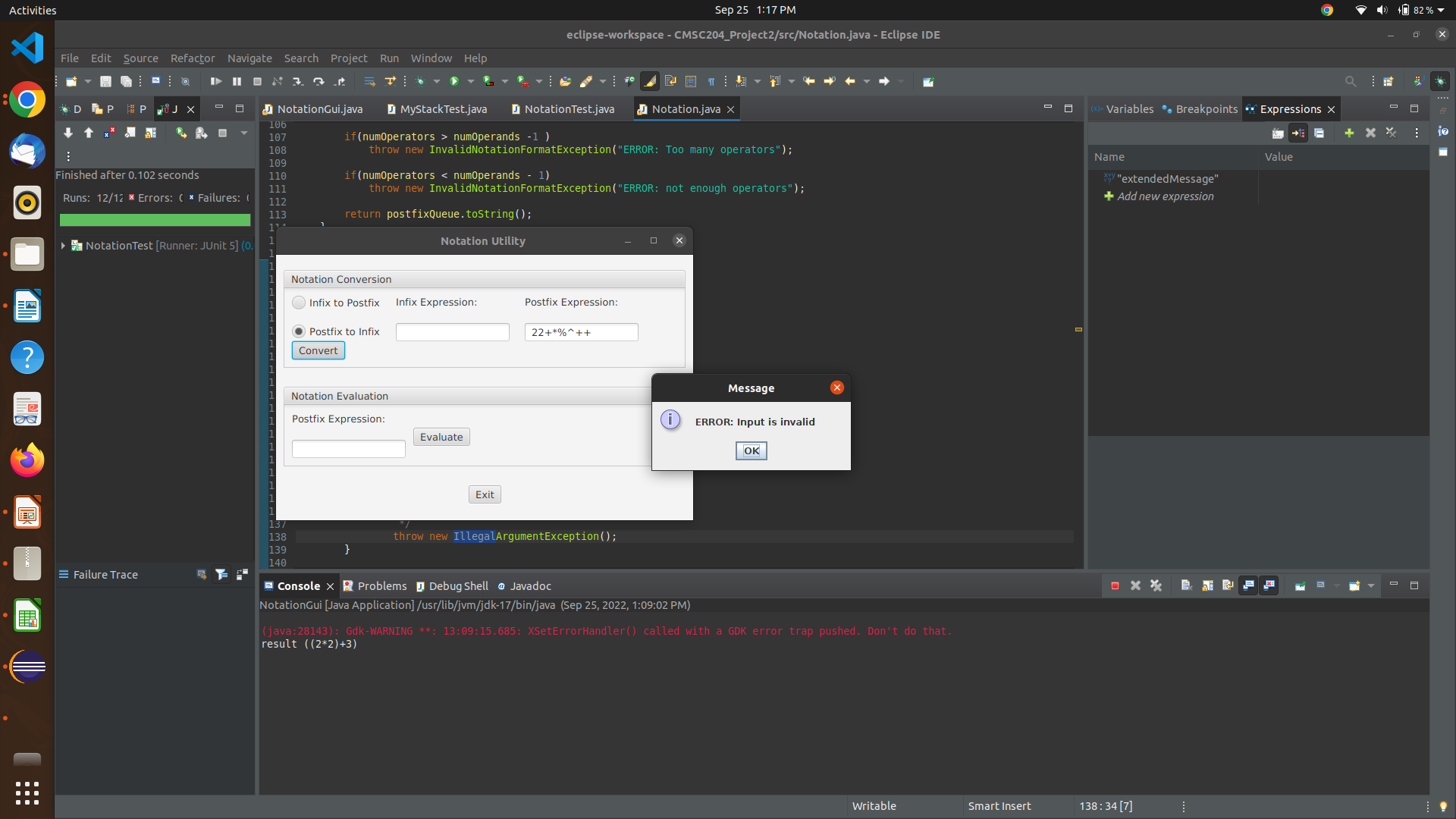
result: InvalidNotationFormatException



case: “22+\*%^++”

expected: InvalidNotationFormatException

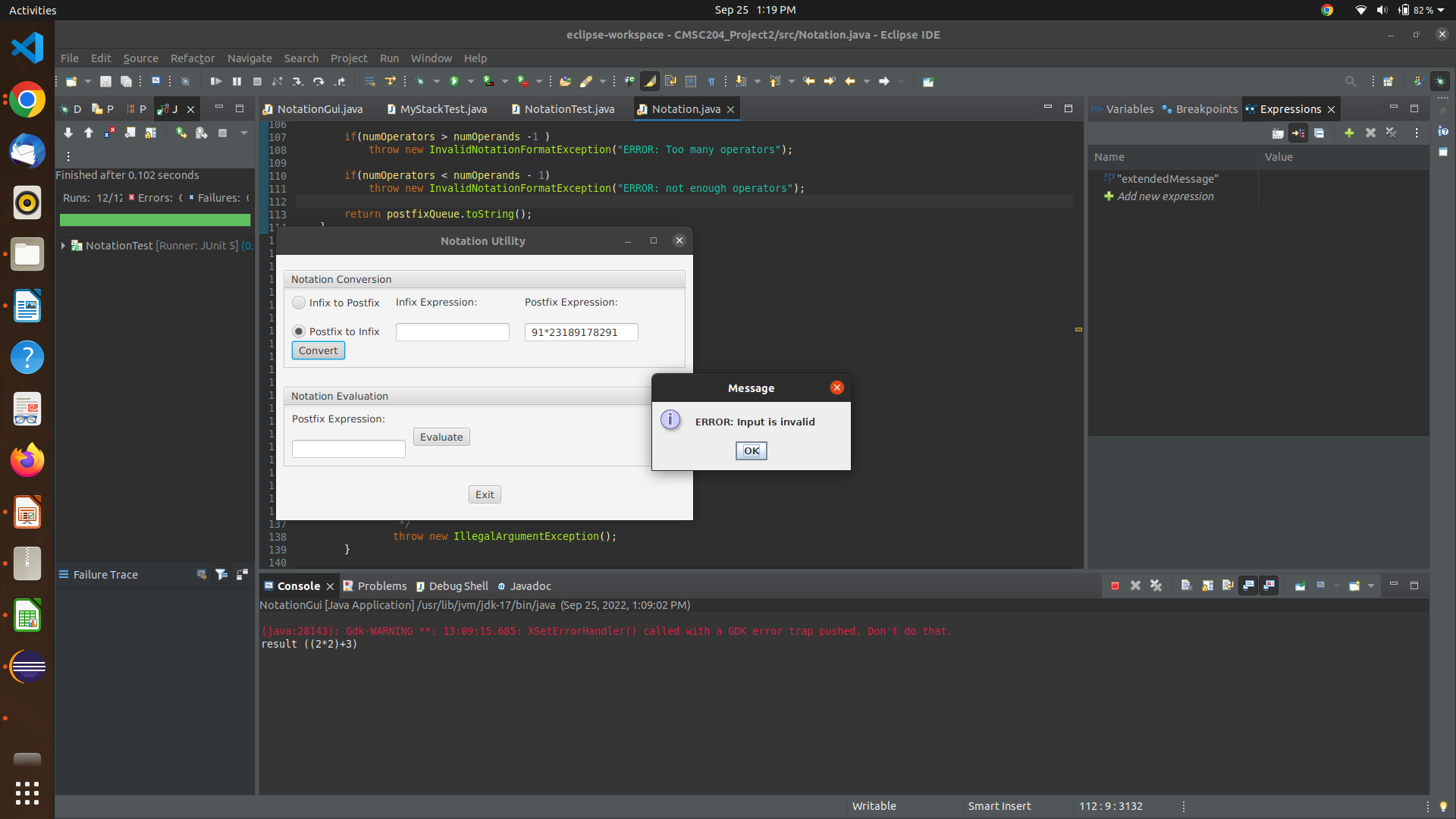
result: InvalidNotationFormatException



case: “91\*23189178291”

expected: InvalidNotationFormatException

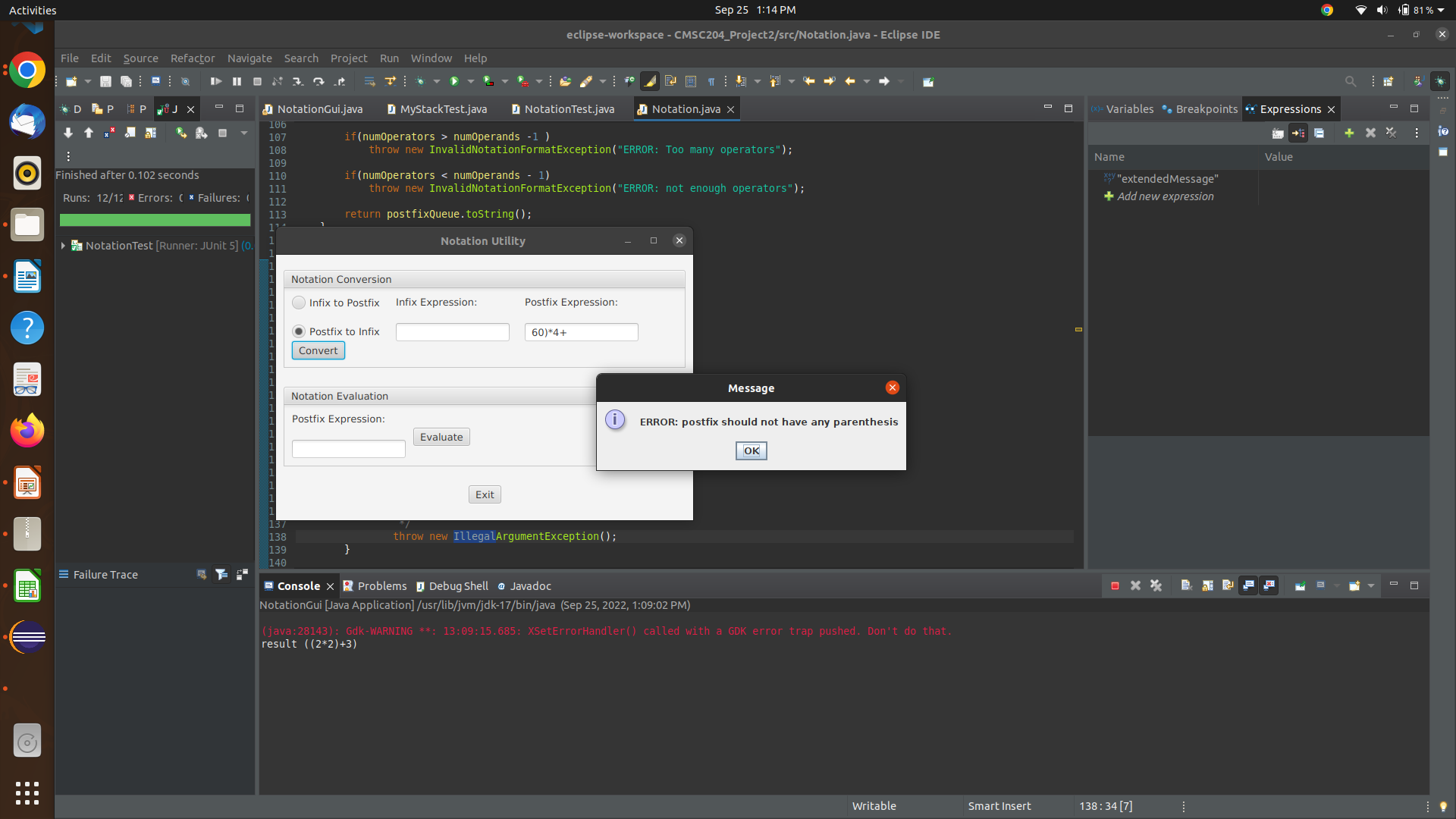
result: InvalidNotationFormatException



case: “60)\*4+”

expected: InvalidNotationFormatException

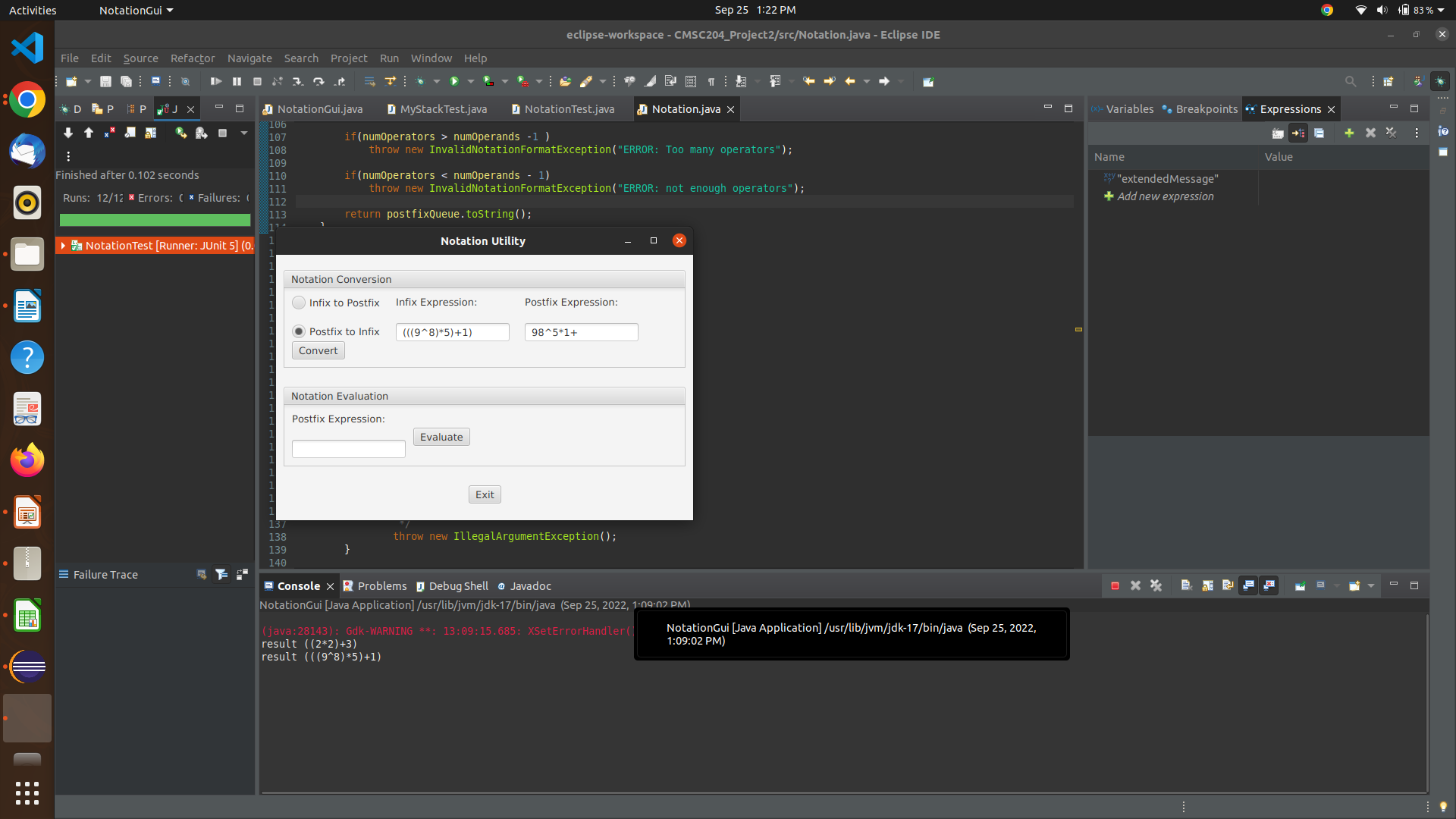
result: InvalidNotationFormatException



case: “98^5\*1+”

expected: “(((9^8)\*5)+1)”

result: “(((9^8)\*5)+1)”

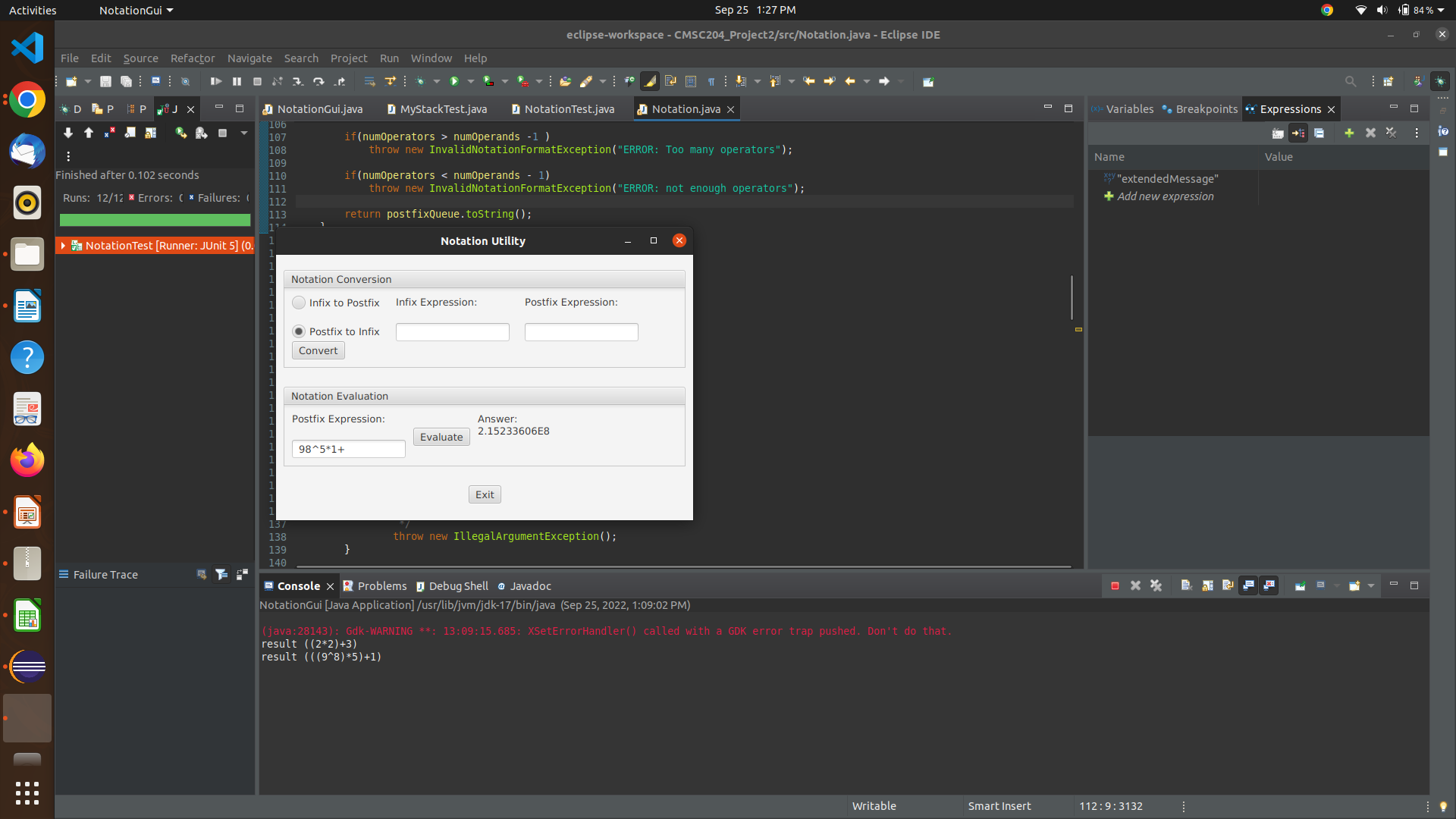


**Test cases (evaluate postfix method):**

case: “98^5\*1+”

expected: “215233606”

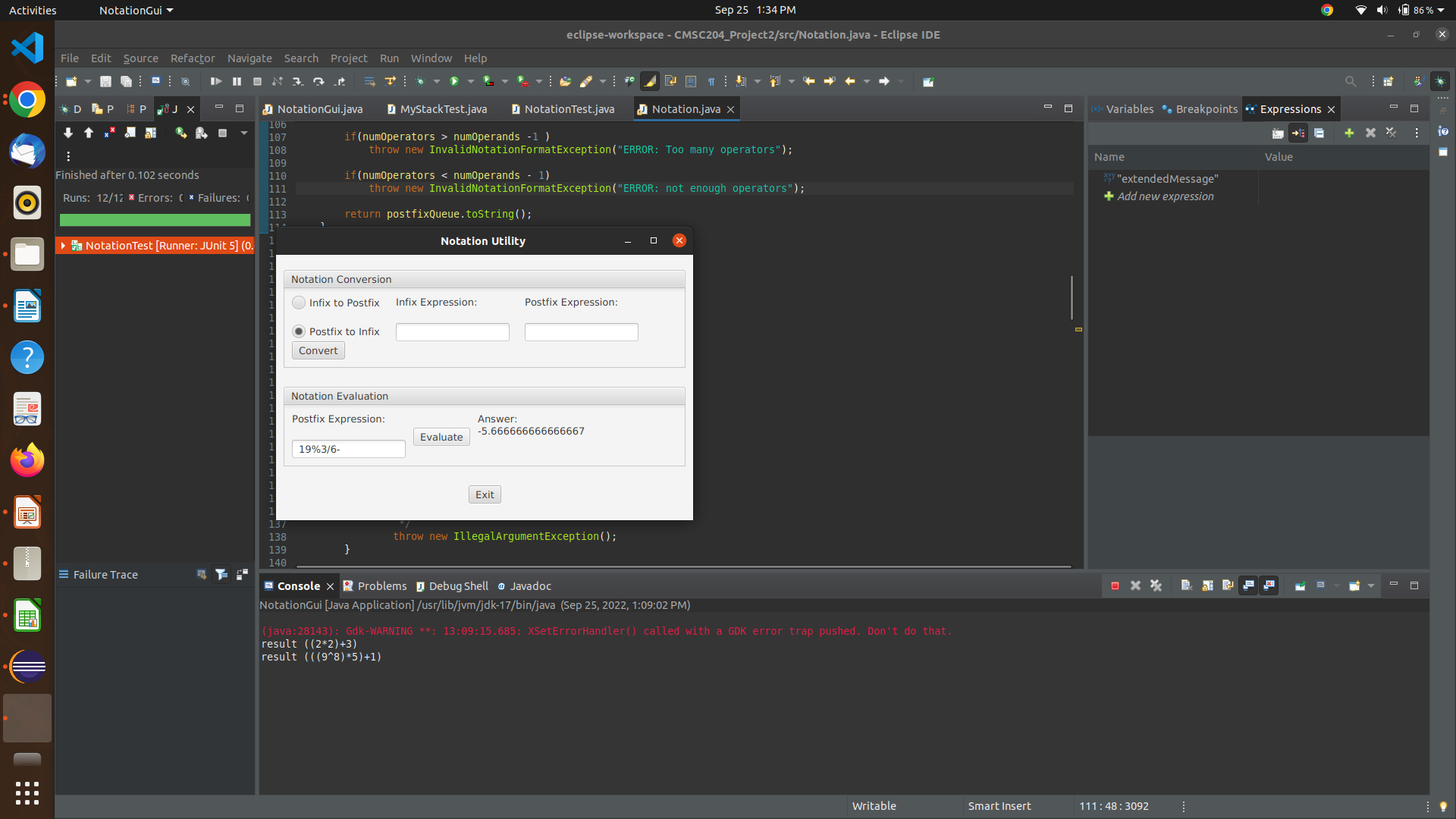
result: “2.15233606E8” (this is the same as expected)



case: “19%3/6-”

expected: “-5.666666667”

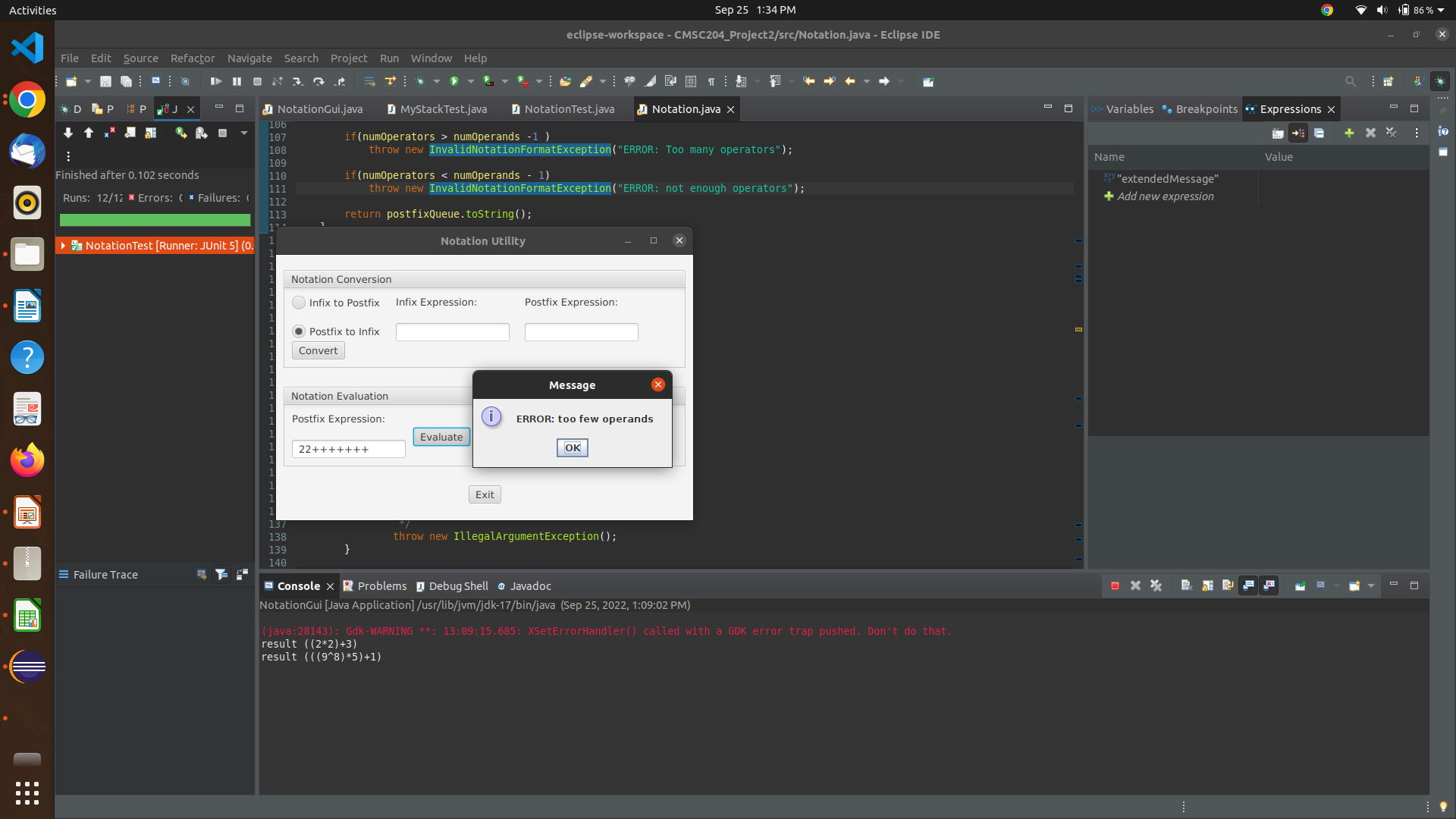
result: “-5.666666667”



case: “22+++++++”

expected: InvalidNotationFormatException

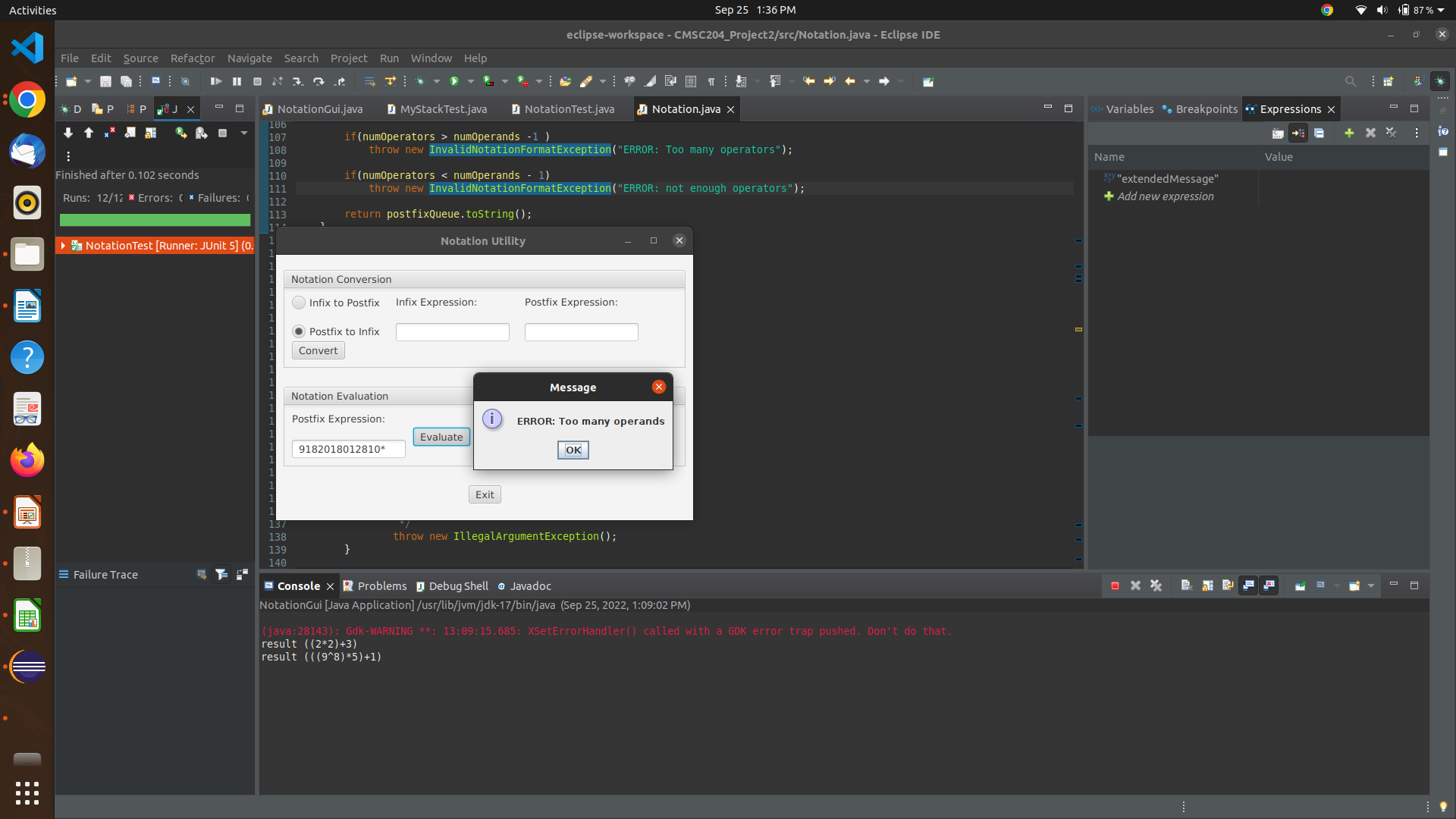
result: InvalidNotationFormatException



case: ”9182018012810\*”

expected: InvalidNotationFormatException

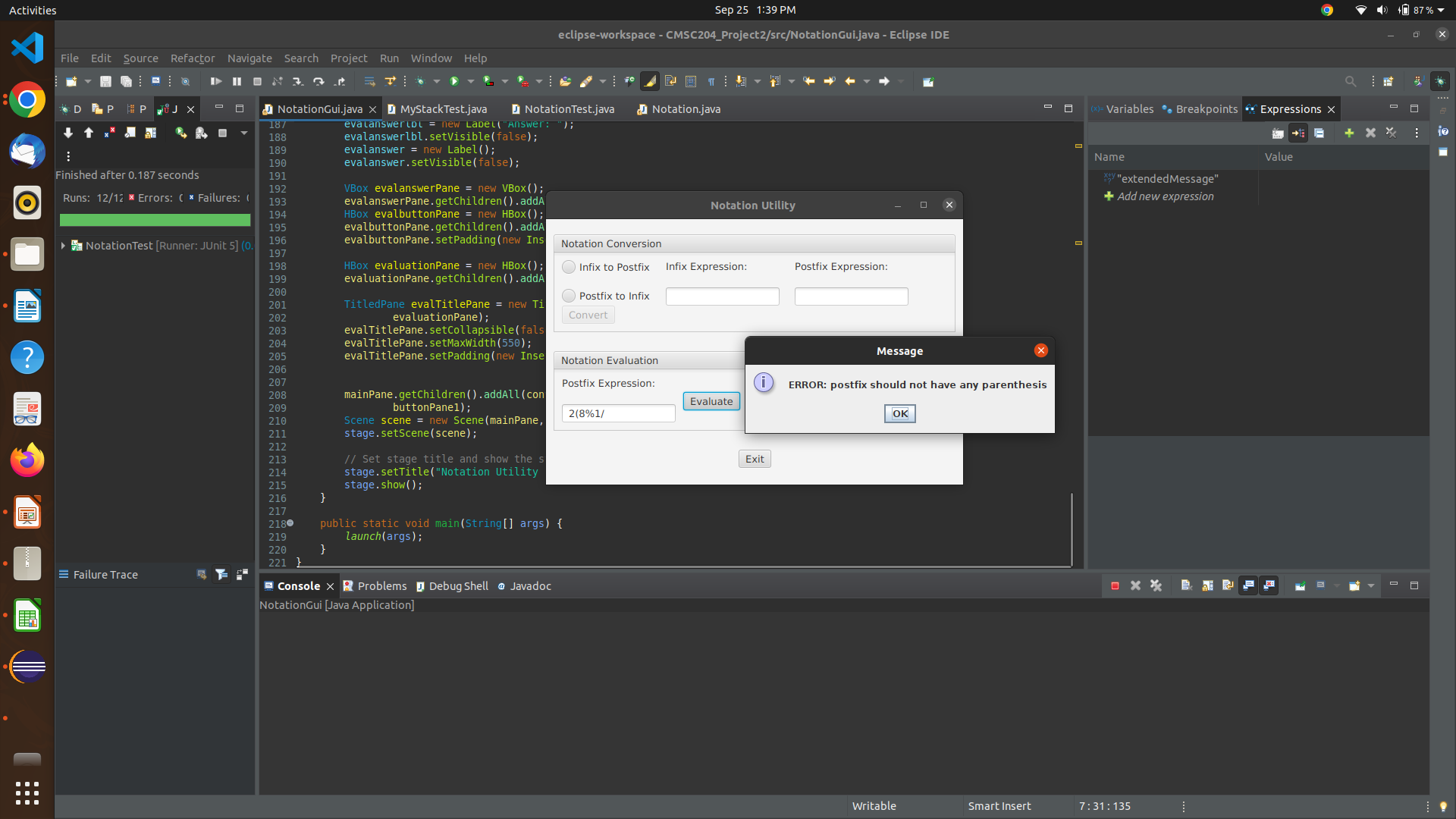
result: InvalidNotationFormatException



case: “2(8%1/”

expected: InvalidNotationFormatException

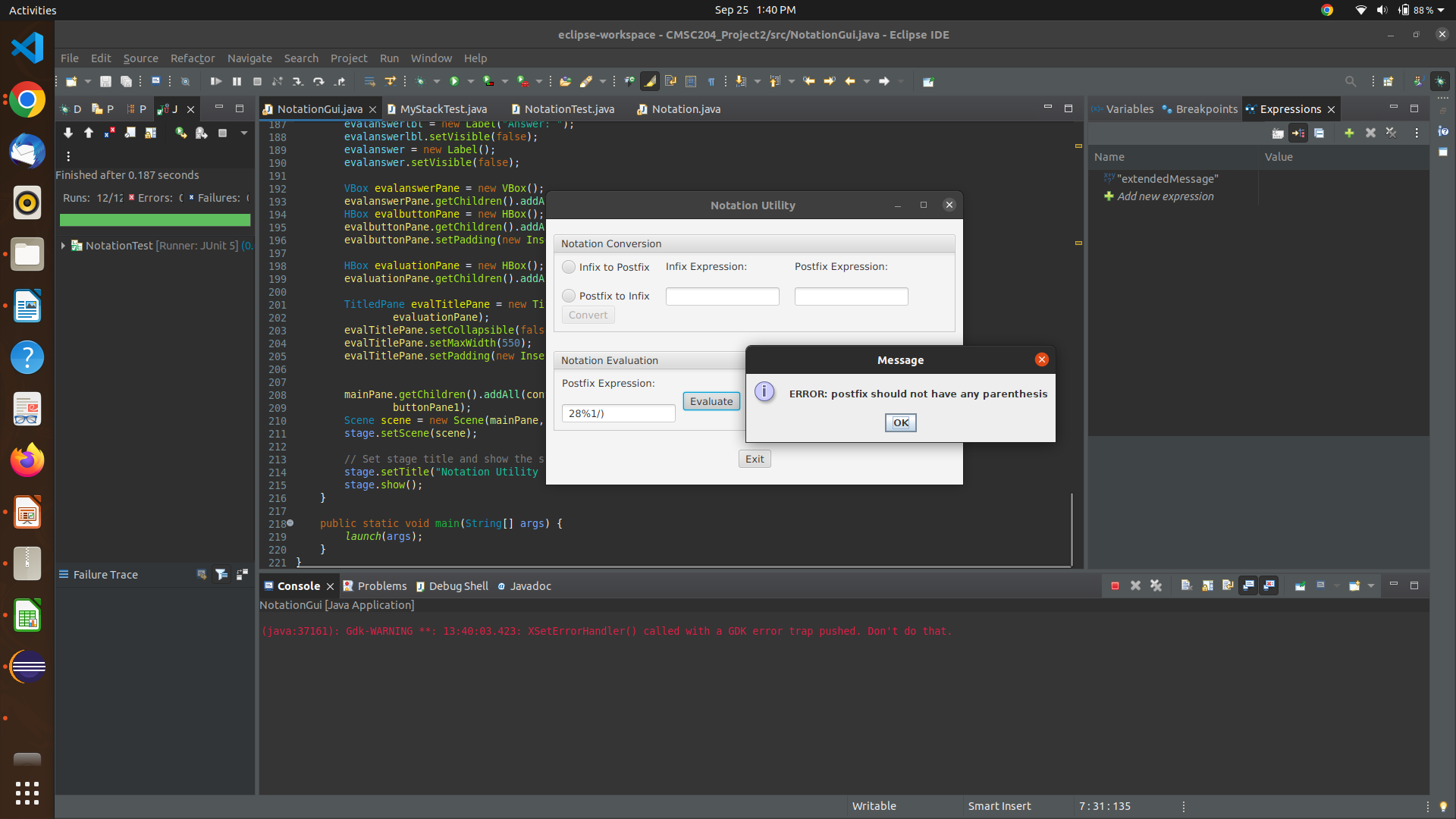
result: InvalidNotationFormatException



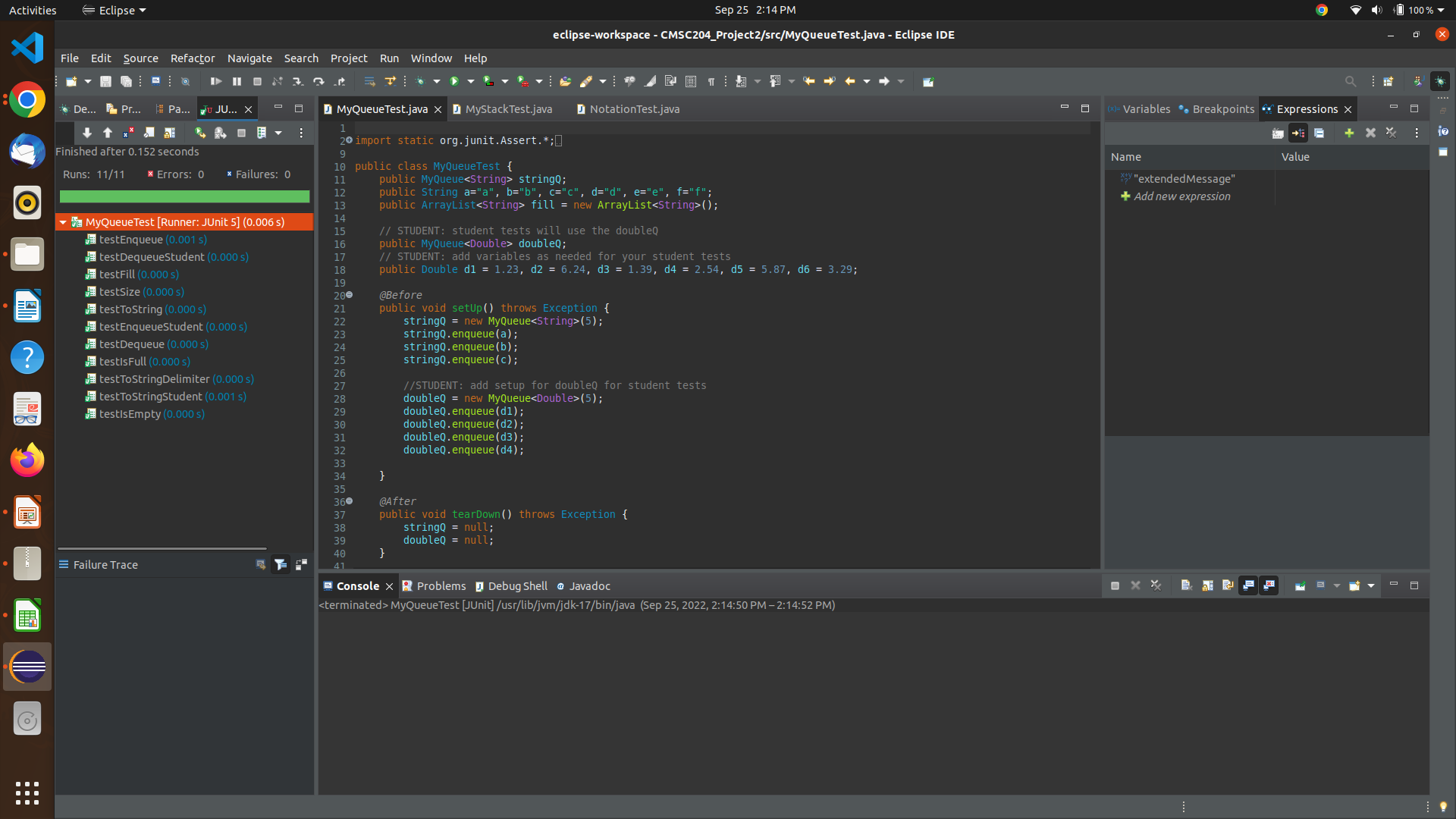
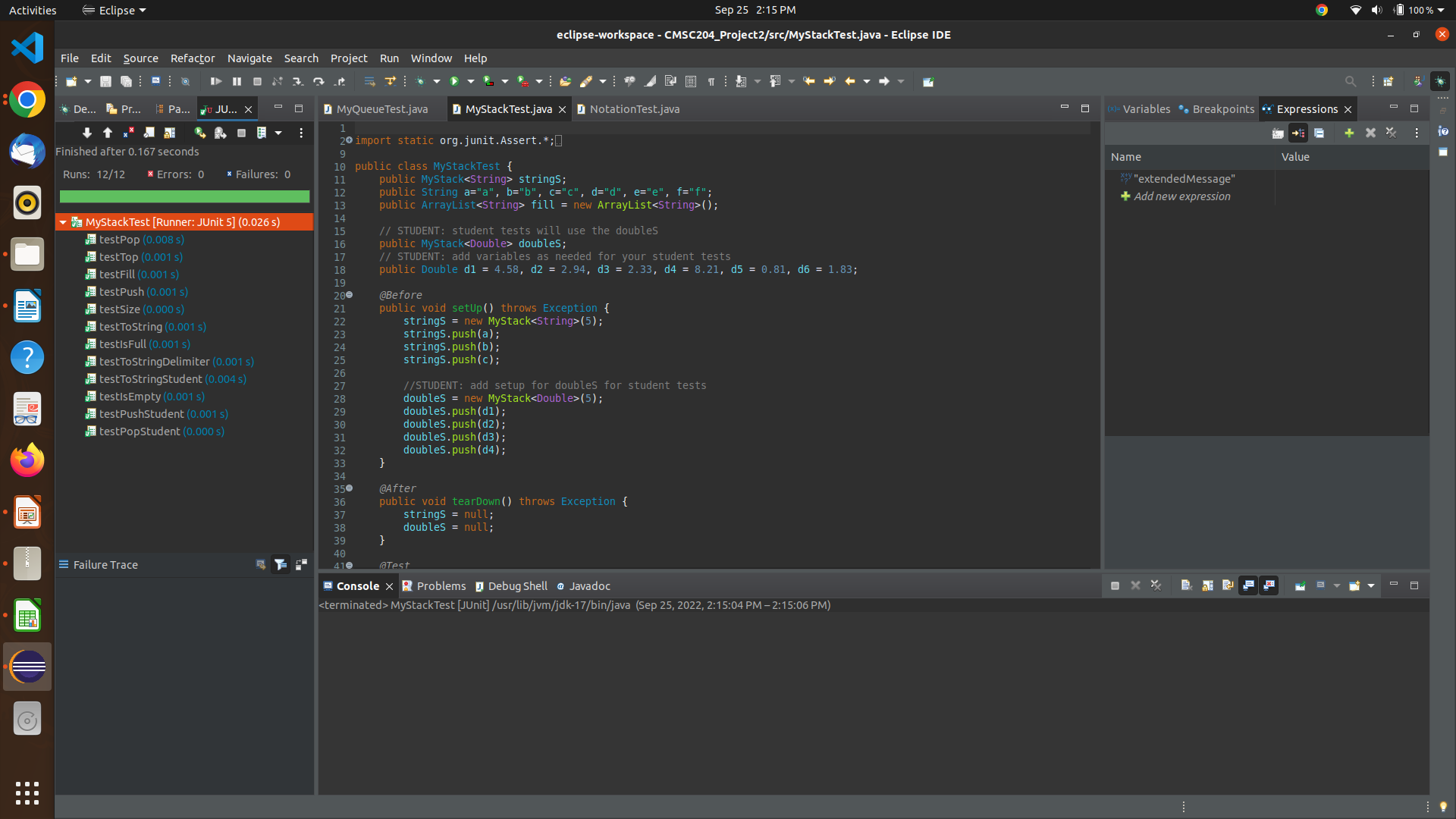
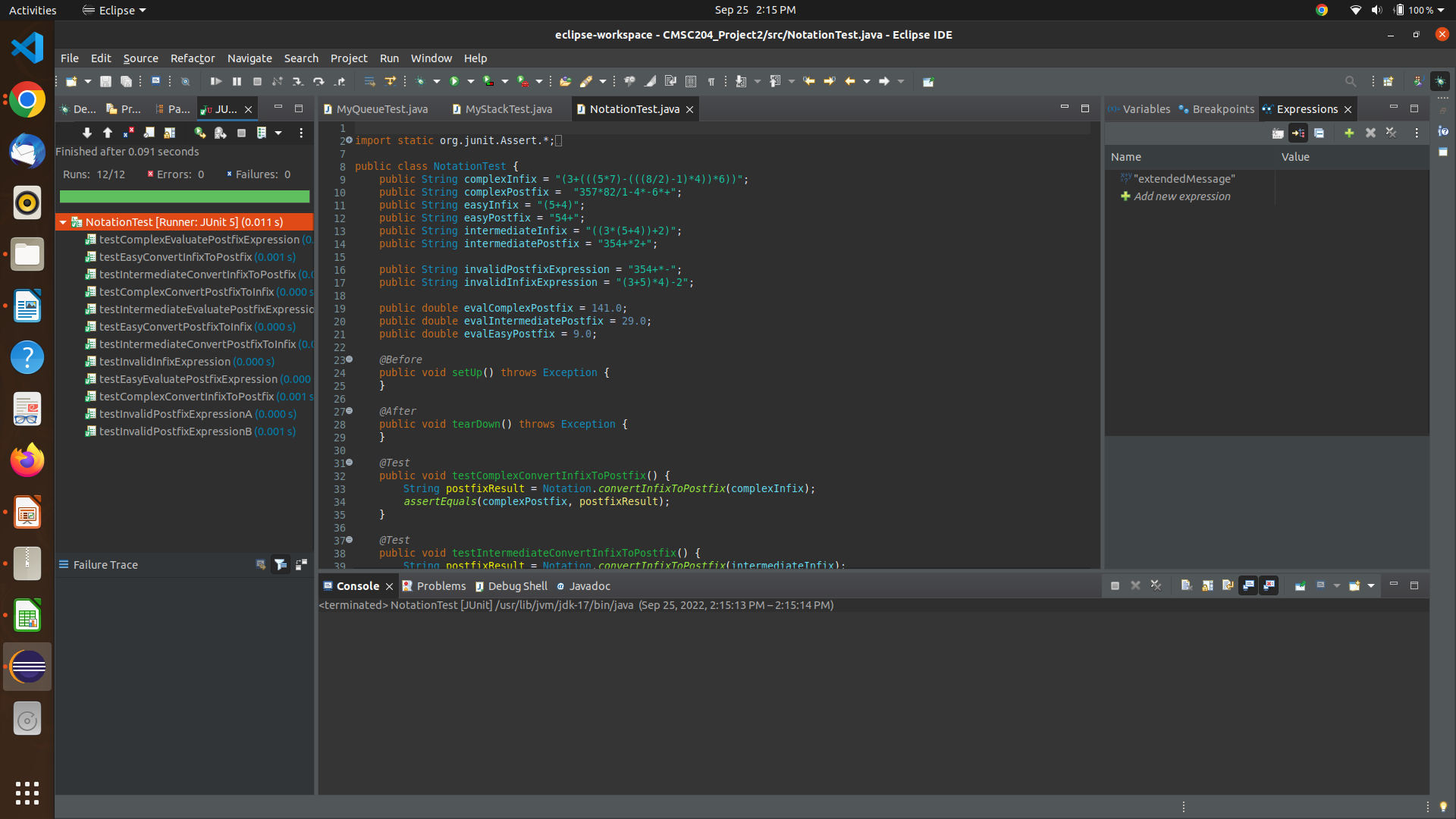
case: “28%1/)”

expected: InvalidNotationFormatException

result: InvalidNotationFormatException



**Screenshots of JUnit Tests:**

****

**Github screenshot:**