Operator precedence

Before exploring the Reverse Polish Notation (RPN), it is important to understand operator precedence and why it matters. Operator precedence refers to the rules that determine which operations are carried out first in an infix mathematical expression. For example, parentheses are done first, followed by exponents, then multiplication and division (which have the same level of precedence), and then addition and subtraction (which also have the same level). These rules ensure that an expression like 1 + 2 × 3 produces the correct result without needing extra brackets. While this form works well for humans as it's easy to read, it creates additional complexity for computers. A computer cannot simply calculate an infix written expression from left to right, it must first read the entire expression, check the precedence of each operator, and rearrange or group parts of it before performing the actual calculations. This makes the evaluation process less straightforward and requires extra processing steps. Reverse Polish Notation (RPN), also known as postfix notation, offers a solution to this problem by structuring expressions in a way that removes the need for precedence rules within the calculation itself (Ada Computer Science, 2025).

What is RPN?

Reverse Polish Notation (RPN) is a way of writing mathematical expressions where the operator comes after the operands. For example, instead of writing 1 + 2 as in standard infix notation, you'd write 1 2 + in RPN. Reverse Polish Notation removes the need for operator precedence rules within the expression itself. The order of operations is already built into how the expression is written (Ada Computer Science, 2025). This allows a computer to process and calculate the expression step by step as it reads it by using a stack data structure. Each time an operator appears, it directly applies it to the last two operands that were popped from the stack. RPN can make expression evaluation more straightforward and efficient for computers, since there is no need to repeatedly check precedence rules or manage brackets during calculation.

When you're dealing with more than two operands, the order in which operators are applied becomes a bit tricky to write in the RPN notation. In infix notation, we rely on operator precedence rules (like multiplication or division before addition) to ensure the calculation is performed correctly. When converting to Reverse Polish Notation (RPN), this precedence influences the way you write the equation in RPN for example - if you were to write $2 \times 3 + 4$ as $2\ 3\ 4 \times +$ that would be incorrect as when the multiplication operator is popped it performs it on the two operands popped before it so it does $3 \times 4 + 2$ instead. To fix this you'd need to keep the order of precedence in mind and then write the appropriate RPN form which is $2\ 3 \times 4 +$.

Exploring a more complicated infix expression and writing it in RPN form

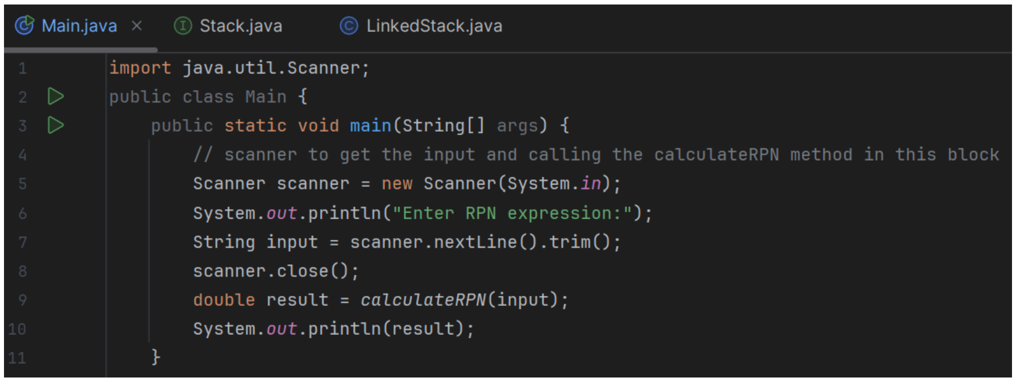
Lets say we want to write (3^2+4)*4*(3+2) in RPN form,

To convert this expression to RPN, we first need to calculate the parts inside the brackets because parentheses have the highest precedence.

In the first bracket (3^2 + 4) 3^2 can be written as,
3 2 ^
and then we add on 4 which makes it,
3 2 ^ 4 +
and then multiply this by 4 so,
3 2 ^ 4 + 4 *
Then work out 3 and 2,
3 2 +
So the final expression will be,
3 2 ^ 4 + 4 * 3 2 + *

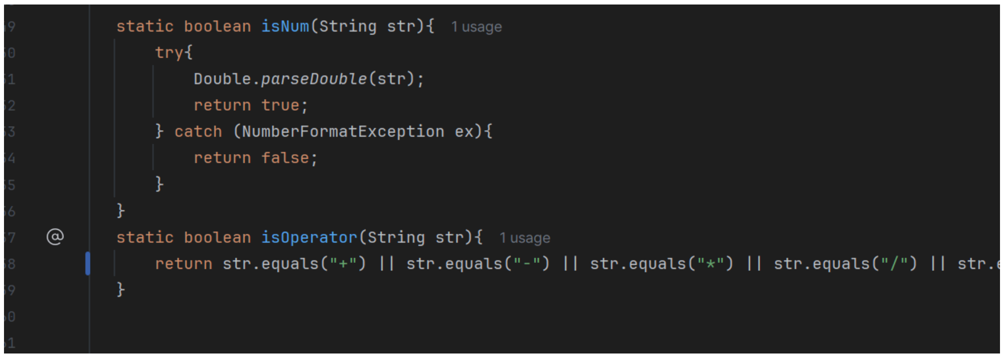
Extension

I started working on the program by importing the “Stack” interface and “LinkedStack”class from the github repository and then created a Main class where I worked on the functionality.



```java
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        // scanner to get the input and calling the calculateRPN method in this block
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter RPN expression:");
        String input = scanner.nextLine().trim();
        scanner.close();
        double result = calculateRPN(input);
        System.out.println(result);
    }
```

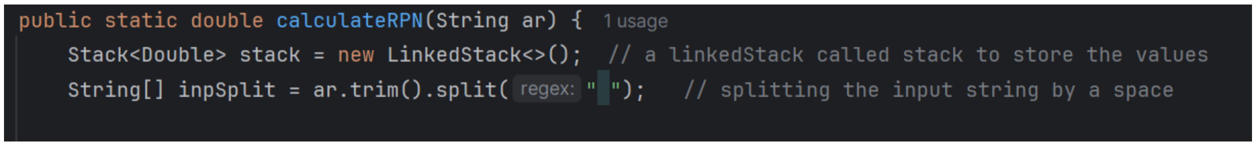
I wanted to take input from the user by using a scanner object so I imported the scanner and in the constructor method created a string called input that holds the expression the scanner takes and a double result.



```java
static boolean isNum(String str){
    try{
        Double.parseDouble(str);
        return true;
    } catch (NumberFormatException ex){
        return false;
    }
}
static boolean isOperator(String str){
    return str.equals("+") || str.equals("-") || str.equals("*") || str.equals("/") || str.
}
```

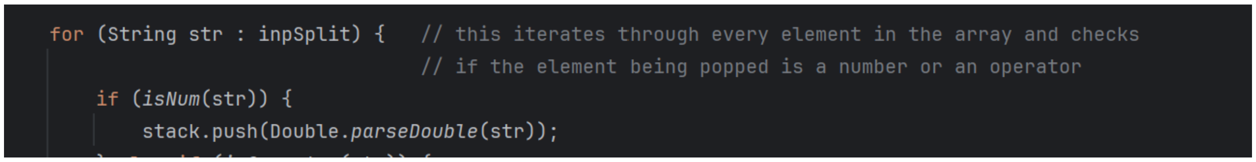
I made three methods: calculateRPN(), isNum() and isOperator(). isNum() and isOperator() methods are to check if the element is a number or operator and if it's a number it is then parsed as a double and pushed onto the stack(this happens in the ”calculateRPN()” method).

Now onto the Main method, I declare and initialize a stack called “stack” that holds Double objects. This is possible because of the generic T used in the stack interface and LinkedStack class. I also declare an string array called inpSplit that splits the string taken from the user separated by a space and adds them onto the array as single elements.



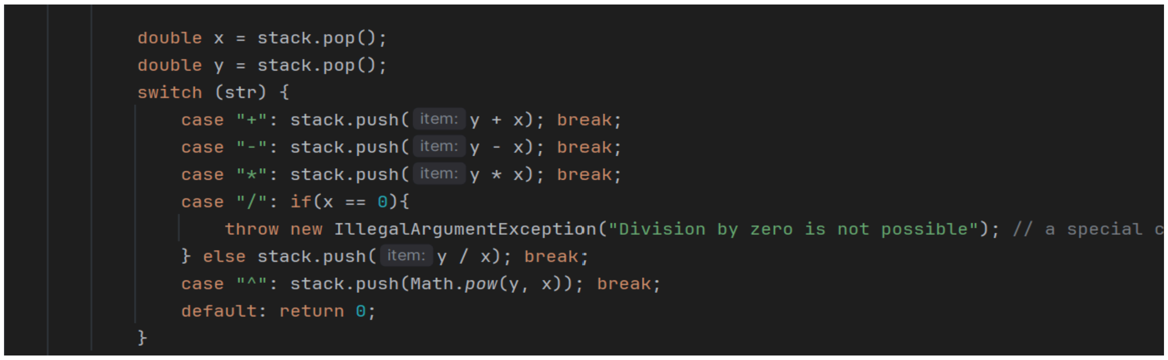
```
public static double calculateRPN(String ar) {  1 usage
    Stack<Double> stack = new LinkedStack<>();  // a linkedStack called stack to store the values
    String[] inpSplit = ar.trim().split( regex: " ");   // splitting the input string by a space
```

After declaring the array I made an enhanced for loop to iterate through the array and check if the element is a number or an operator using the isNum() and isOperator methods or if it's something else(in this case I made it so that it throws an exception).



```
  for (String str : inpSplit) {   // this iterates through every element in the array and checks
                                  // if the element being popped is a number or an operator
      if (isNum(str)) {
          stack.push(Double.parseDouble(str));
```

If the element is a number it is converted into a double and pushed to the stack. If the element is an operator it pops the last two numbers, stores them in variables x and y, and then goes through a switch for every operator case shown below;

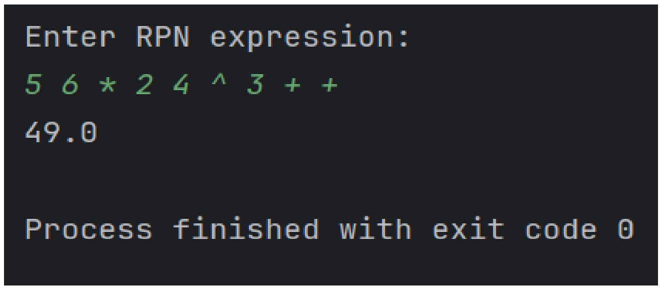


```
            double x = stack.pop();
            double y = stack.pop();
            switch (str) {
                case "+": stack.push( item: y + x); break;
                case "-": stack.push( item: y - x); break;
                case "*": stack.push( item: y * x); break;
                case "/": if(x == 0){
                    throw new IllegalArgumentException("Division by zero is not possible"); // a special c
                } else stack.push( item: y / x); break;
                case "^": stack.push(Math.pow(y, x)); break;
                default: return 0;
            }
```

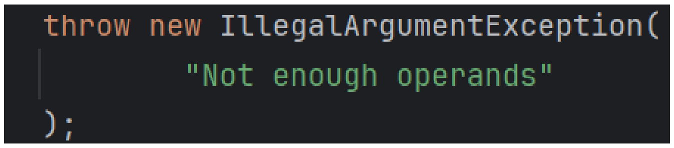
Note - *the exceptions were added after the initial draft of the code but the screen shots are of the finished code. I’ll elaborate on the validation later on.*

The program was now functional and I decided to test it.

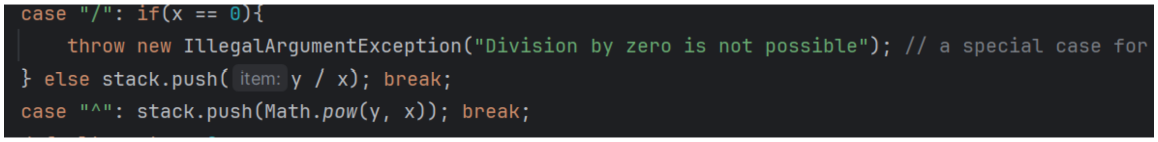
First test (the same one from truth table)


```
Enter RPN expression:
5 6 * 2 4 ^ 3 + +
49.0

Process finished with exit code 0
```

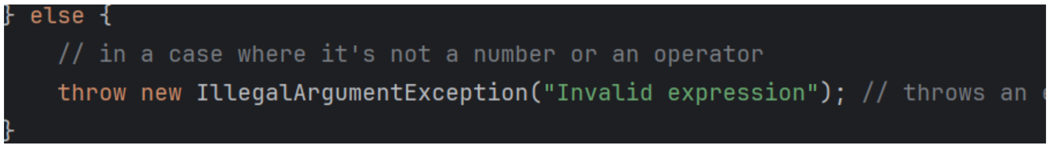
The calculation was correct, and this was the initial build I uploaded to GitHub. But at that stage, there was no input validation. I later added exceptions to handle common errors, invalid input (for example, a regular string instead of an RPN expression), division by zero, and cases where there aren't enough operands (for example, 2 +).


```
throw new IllegalArgumentException(
        "Not enough operands"
);
```

*exception for not enough operands (GeeksforGeeks, 2016).


```
case "/": if(x == 0){
    throw new IllegalArgumentException("Division by zero is not possible"); // a special case for
} else stack.push( item: y / x); break;
case "^": stack.push(Math.pow(y, x)); break;
```

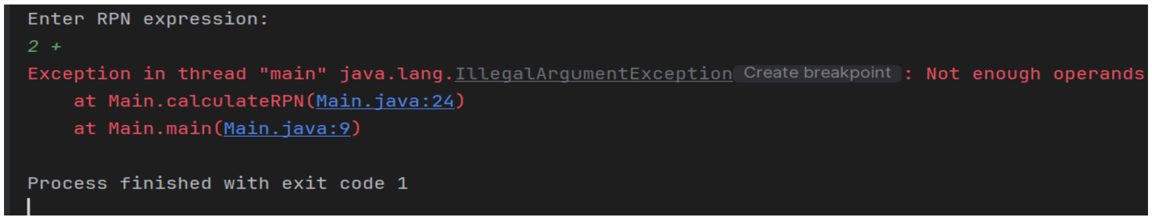
*division by zero (GeeksforGeeks, 2016).


```
} else {
    // in a case where it's not a number or an operator
    throw new IllegalArgumentException("Invalid expression"); // throws an
}
```

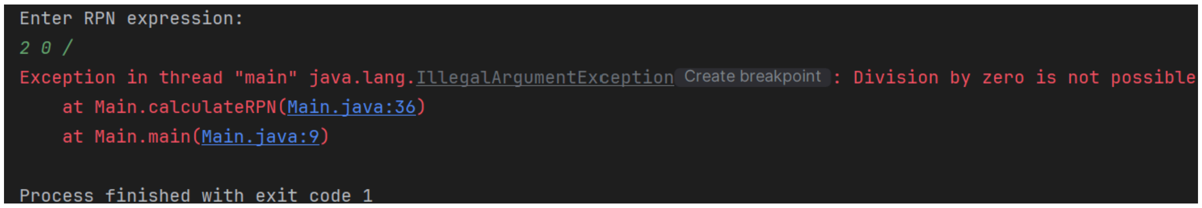
*invalid expression for when the input is not a number or an operator (GeeksforGeeks, 2016).

Testing all the exception cases

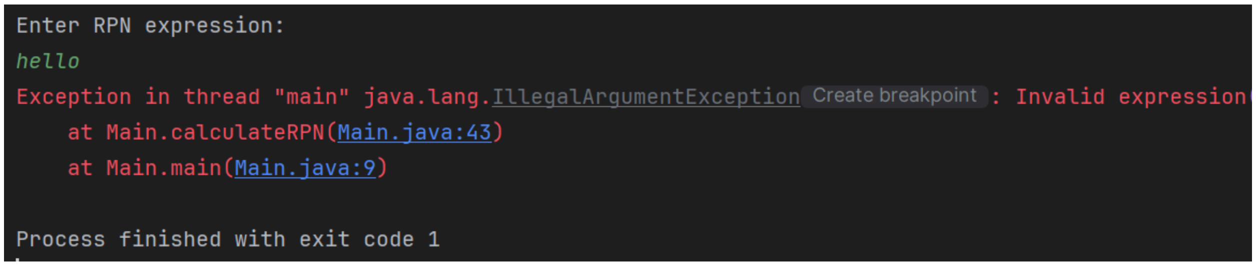
Case 1 with not enough operands:


```
Enter RPN expression:
2 +
Exception in thread "main" java.lang.IllegalArgumentException Create breakpoint : Not enough operands
    at Main.calculateRPN(Main.java:24)
    at Main.main(Main.java:9)

Process finished with exit code 1
```

Case 2 trying to divide by 0:


```
Enter RPN expression:
2 0 /
Exception in thread "main" java.lang.IllegalArgumentException Create breakpoint : Division by zero is not possible
    at Main.calculateRPN(Main.java:36)
    at Main.main(Main.java:9)

Process finished with exit code 1
```

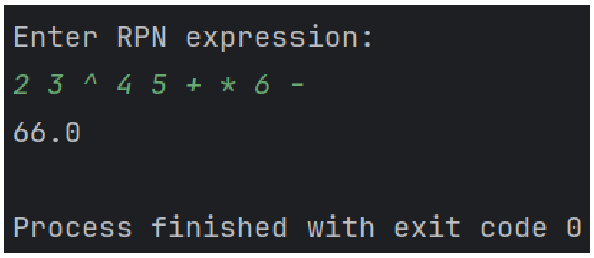
Case 3 with an invalid expression:



```
Enter RPN expression:
hello
Exception in thread "main" java.lang.IllegalArgumentException Create breakpoint : Invalid expression
    at Main.calculateRPN(Main.java:43)
    at Main.main(Main.java:9)

Process finished with exit code 1
```

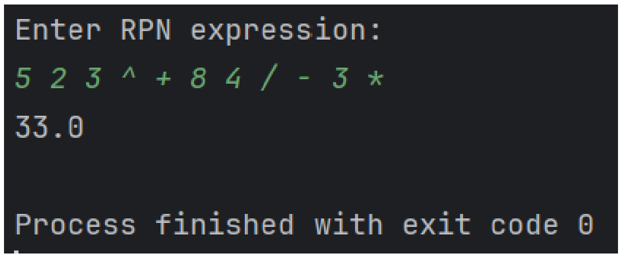
Testing more complex RPN expressions:

Expression 1: 2 3 ^ 4 5 + * 6 -
Result:



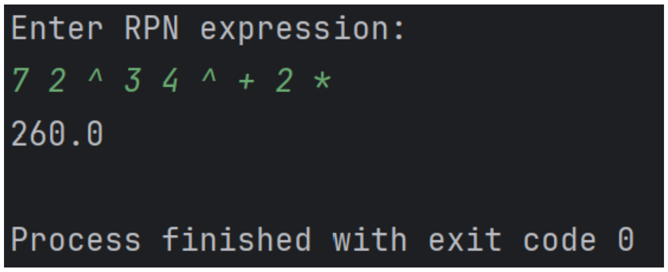
```
Enter RPN expression:
2 3 ^ 4 5 + * 6 -
66.0

Process finished with exit code 0
```

Expression 2: 5 2 3 ^ + 8 4 / - 3 *
Result:



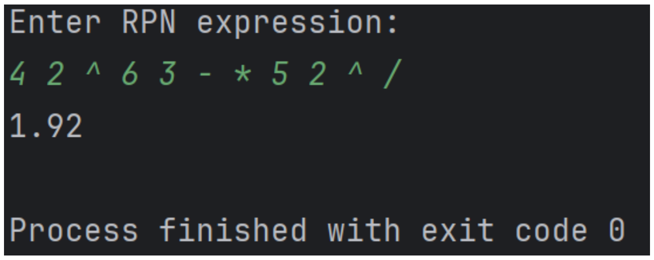
```
Enter RPN expression:
5 2 3 ^ + 8 4 / - 3 *
33.0

Process finished with exit code 0
```

Expression 3: 7 2 ^ 3 4 ^ + 2 *
Result:



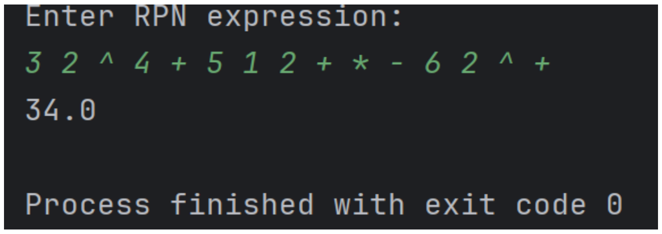
```
Enter RPN expression:
7 2 ^ 3 4 ^ + 2 *
260.0

Process finished with exit code 0
```

Expressions 4: 4 2 ^ 6 3 - * 5 2 ^ /
Result:



```
Enter RPN expression:
4 2 ^ 6 3 - * 5 2 ^ /
1.92

Process finished with exit code 0
```

Expression 5: 3 2 ^ 4 + 5 1 2 + * - 6 2 ^ +
Result:



```
Enter RPN expression:
3 2 ^ 4 + 5 1 2 + * - 6 2 ^ +
34.0

Process finished with exit code 0
```

References-

- Wikipedia. (2020). Reverse Polish notation. [online] Available at: https://en.wikipedia.org/wiki/Reverse_Polish_notation.
- Ada Computer Science. (2025). Ada Computer Science. [online] Available at: https://adacomputerscience.org/concepts/trans_rpn.
- GeeksforGeeks (2016). Java Exception Handling. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/java/exceptions-in-java/.