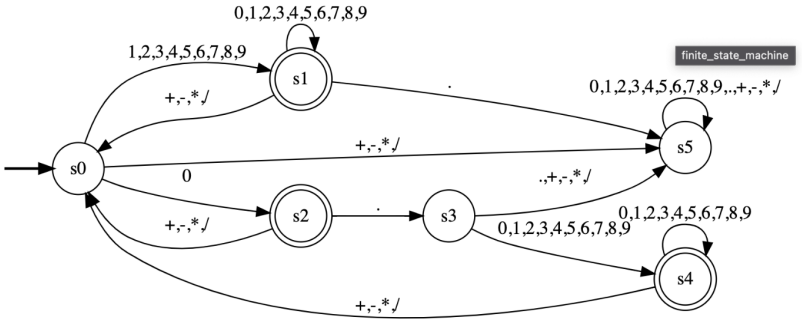
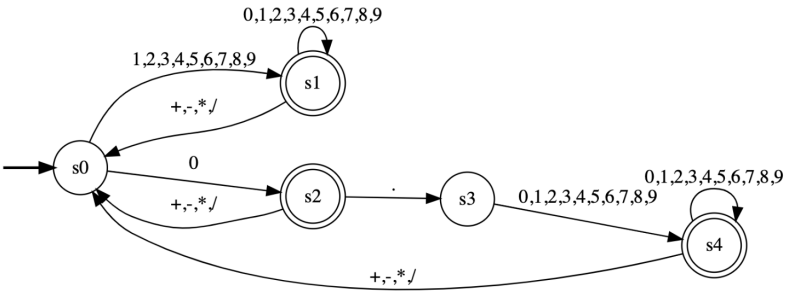
From the specifications given for this assignment, I interpreted the accepted inputs to be, any valid number by itself, this includes any integer, 0, and any decimal number that starts with 0 and has valid integer/s following (e.g. 0.12) including 0.0. I decided to ignore whitespaces unless they were in between two integers (e.g. 5 6 + 2) then that would present an invalid input. From these specifications, this is my formal Specification description.

Q = {s0,s1,s2,s3,s4,s5}
Σ = {0-9, . ,+,-,/,*}
δ = Q x Σ → Q
F = {s1,s2,s4}
s0 ∈ Q is start state

This was my first DFA that I made following these specifications. This included s5 which was a state that would enter a terminate loop if the wrong input was entered.



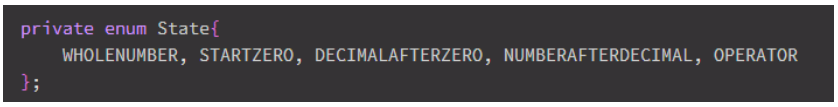
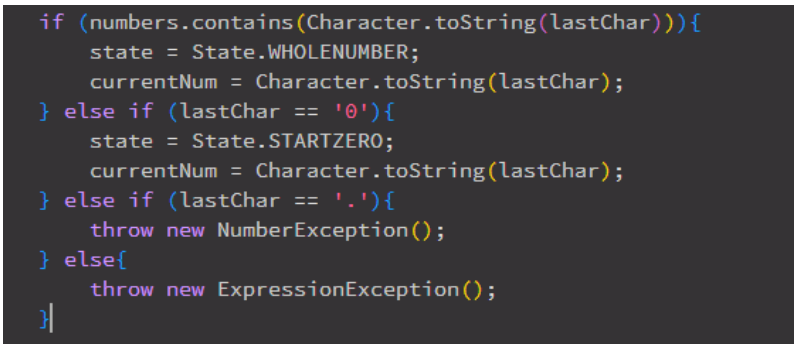
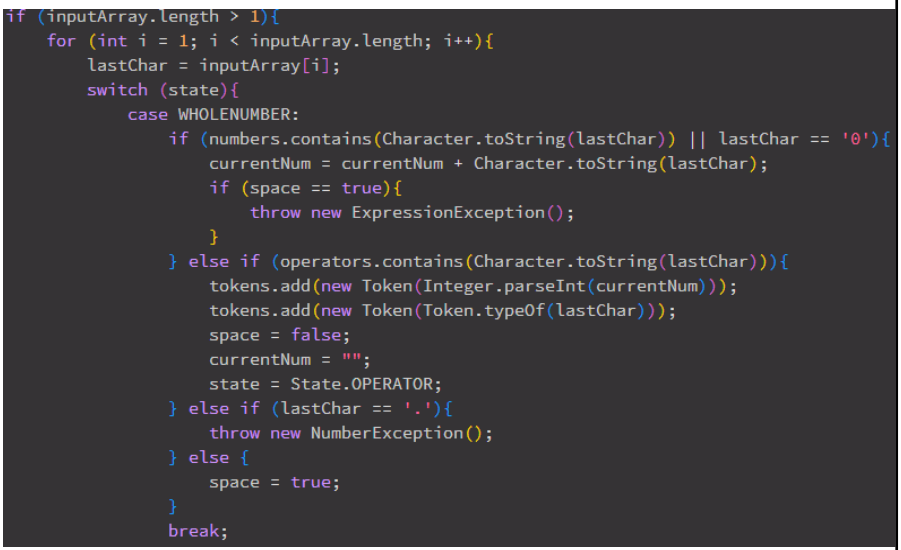
However, when making the code for this, entering an invalid input and going into a terminate loop did not make sense as it would keep reading the inputs even though it was a guaranteed invalid input. Thus, when developing the DFA in Java, I removed s5 and made it throw exceptions as soon as an invalid input was given. This would reduce the runtime for errors and immediately exit the loop no matter the amount of inputs. This resulted in basing it off a DFA more like the one below.



The start state, s0, in the program is called the OPERATOR state. This state would throw an exception for any first input that was not an integer, from this state it can go to s1, called WHOLENUMBER, or s2, called STARTZERO. These states would determine the starting integer of the number. If the number started with 0 then it has to be followed by a decimal which would take it to s3, DECIMALAFTERZERO. As ending on a decimal would result in a number exception, it needs to be followed up with another integer, This would move it to s4, NUMBERAFTERDECIMAL, which is an accept state. If from s1,s2 or s4, an operator is

given, it will transition to s0, where another integer would need to be given in order to make it a valid input.

After deciding to do my code in Java, I was able to implement finite automation by making the input string into an array of characters and looping through that array testing whether the input is 0, 1-9, an operator, or a decimal. Depending on what the input is, it switches to the appropriate state and eventually adds the tokens to a list as the output. However I believe I could have improved my code as it was very rushed, I do not think much could have been done to improve the ability for this program to act as a finite state machine.

	I started by creating a list of enumerations, which would act as the states that I would switch between
	This block of code is run before the loop which will decide the state based on the first character. As there are only two states that the start state can move from, there are only two conditions that will change the state. I initially had it so anything besides a number from 0-9 would immediately be an invalid input, but I had to decide whether it was a number exception or an expression exception so the last two else statements were added.
	This block of code now implements a loop where each character of the inputArray is being checked (if the array is more than one). It will set the lastChar to the character. Then, it will enter a switch statement where it decides what to do based on its current state. This part shows the WHOLENUMBER state (q1). It has 4 if statements where it will either add the number to the running currentNum, add the number and operator as a token and switch to the OPERATOR state or throw exceptions. This is the basic layout of how all of my switch statements are. It uses the state it's in and the current input to decide whether to add tokens, switch states or throw exceptions.

```
switch (state){
        case OPERATOR:
            //throw exception, finished with operator
            throw new ExpressionException();
        case DECIMALAFTERZERO:
            throw new NumberException();
        case WHOLENUMBER:
            tokens.add(new Token(Integer.parseInt(currentNum)));
            break;
        case STARTZERO:
        case NUMBERAFTERDECIMAL:
            tokens.add(new Token(Double.parseDouble(currentNum)));
}
```

Finally, after the loop is finished, it will enter the final switch statement where it will decide what to do based on the final state that it was in. If it was not in any of the accept states, WHOLENUMBER, STARTZERO or NUMBERAFTERDECIMAL, an exception will be thrown. However, if it is, it will add the final token.

Some of the challenges while creating my code was making it a true DFA diagram. Initially, when starting to develop the main body I thought it was easier to use a python method called partition() where it can split a string into different parts. I then analysed those parts testing if they were valid or not and then adding them to the array as tokens. After I was nearly completed with this, I realised that it was not finite automation and decided to restart. One problem that I came across while developing this finite state machine, was that I had to ignore spaces when they were before and after operators like “5 + 4” but I also had to keep track of spaces as the input “5 6 + 6” would be invalid. This meant I could not just ignore spaces entirely and had to decide if the input was valid or not based on the last input and the one after that. Finally, as I know python and java to similar levels I had trouble deciding which language I wanted to use. I initially completed 13/14 tasks in python as it seemed easier but after looking over the code and the way it was structured I decided to end up completing the assignment in Java and that was found to be much easier for me.