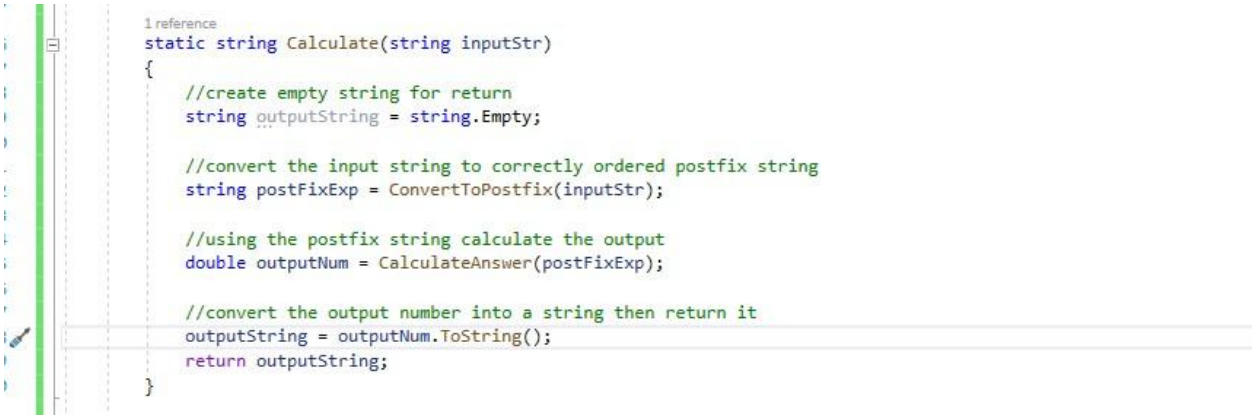
Jagex Programming Text Documentation

Question 1

To begin, I created the main calculation method that is the focus of the test. It would take in a string input in the form of an expression and return the answer also in the form of a string.



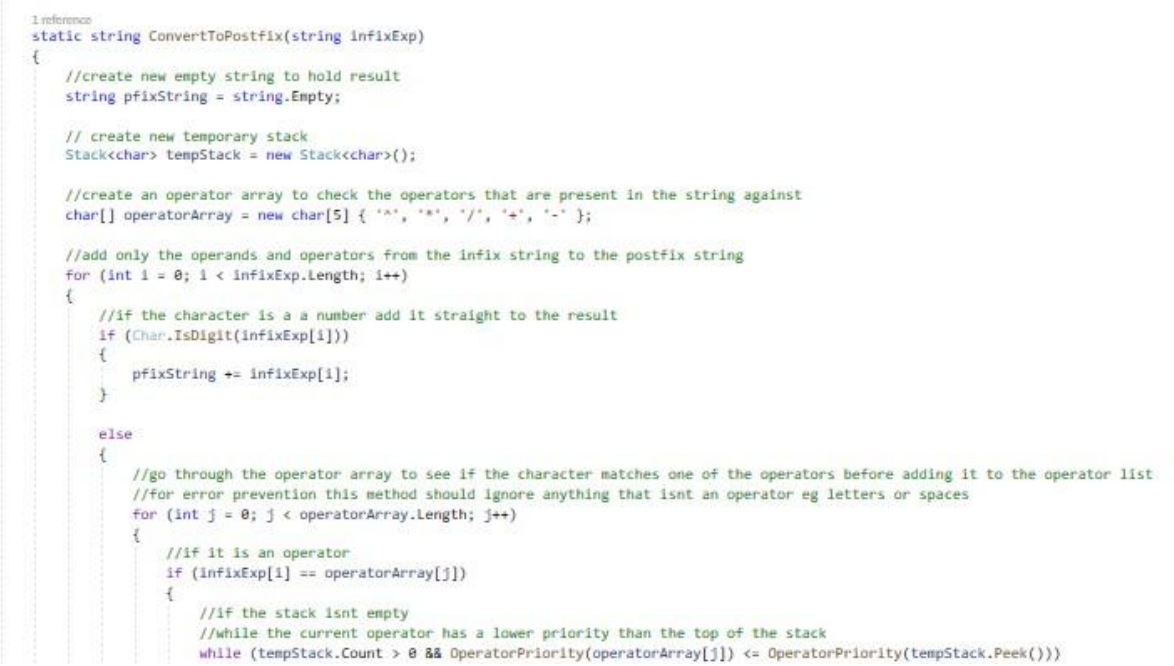
```
1 reference
static string Calculate(string inputStr)
{
    //create empty string for return
    string outputString = string.Empty;

    //convert the input string to correctly ordered postfix string
    string postFixExp = ConvertToPostfix(inputStr);

    //using the postfix string calculate the output
    double outputNum = CalculateAnswer(postFixExp);

    //convert the output number into a string then return it
    outputString = outputNum.ToString();
    return outputString;
}
```

Taking some advice from the test paper I broke the calculation into two parts converting the expression into a postfix format from its initial infix string format. Before then using that format to make it easier to calculate the answer using a stack.



```
1 reference
static string ConvertToPostfix(string infixExp)
{
    //create new empty string to hold result
    string pfixString = string.Empty;

    // create new temporary stack
    Stack<char> tempStack = new Stack<char>();

    //create an operator array to check the operators that are present in the string against
    char[] operatorArray = new char[5] { '^', '*', '/', '+', '-' };

    //add only the operands and operators from the infix string to the postfix string
    for (int i = 0; i < infixExp.Length; i++)
    {
        //if the character is a a number add it straight to the result
        if (Char.IsDigit(infixExp[i]))
        {
            pfixString += infixExp[i];
        }

        else
        {
            //go through the operator array to see if the character matches one of the operators before adding it to the operator list
            //for error prevention this method should ignore anything that isnt an operator eg letters or spaces
            for (int j = 0; j < operatorArray.Length; j++)
            {
                //if it is an operator
                if (infixExp[i] == operatorArray[j])
                {
                    //if the stack isnt empty
                    //while the current operator has a lower priority than the top of the stack
                    while (tempStack.Count > 0 && OperatorPriority(operatorArray[j]) <= OperatorPriority(tempStack.Peek()))
```

```
                {
                    //add the operator from the top of the stack to the postfix string
                    pfixString += tempStack.Pop();
                }
                //add the current operator to the stack
                tempStack.Push(infixExp[i]);
            }
        }
    }
    // pop all remaining operators from the stack into the postfix string
    while (tempStack.Count > 0)
    {
        pfixString += tempStack.Pop();
    }

    //return the string in its postfix format
    return pfixString;
}
```

I used a method containing a switch case to assign a priority to the different operators for use when converting the string into a postfix format.

```
2 references
static int OperatorPriority(char op)
{
    switch (op)
    {
        case '+':
        case '-':
            return 1;

        case '*':
        case '/':
            return 2;

        case '^':
            return 3;
    }
    return -1;
}
```

This is the method I wrote to calculate the answer to the equation, going through each character in the postfix formatted string, if it was a number I would convert it into an int before putting it on the stack. If it was an operator I would then pull the top two numbers from the stack, complete the corresponding calculation then put the new number back on the stack. I repeated this until only one number was left, the answer.

```
1 reference
static double CalculateAnswer(string postfixExp)
{
    //create new empty string to hold result
    string pfixString = string.Empty;

    // create new temporary stack
    Stack<double> tempStack2 = new Stack<double>();

    //for each character in the string
    for (int i = 0; i < postfixExp.Length; i++)
    {
        //if its a digit
        if (Char.IsDigit(postfixExp[i]))
        {
            //convert it into a int and add to the stack
            var n = (double)Char.GetNumericValue(postfixExp[i]);
            tempStack2.Push(n);
        }
        // If the scanned character is an operator, pop two elements from stack apply the operator then push back
        else
        {
            double n1 = tempStack2.Pop();
            double n2 = tempStack2.Pop();

            switch (postfixExp[i])
            {
                case '+':
                    tempStack2.Push(n2 + n1);
                    break;

                case '-':
                    tempStack2.Push(n2 - n1);
                    break;

                case '/':
                    tempStack2.Push(n2 / n1);
                    break;

                case '*':
                    tempStack2.Push(n2 * n1);
                    break;

                case '^':
                    Math.Pow(n2, n1);
                    break;
            }
        }
    }
    return tempStack2.Pop();
}
```

Question 2 and 3

The previous solution worked great for any formula that contained only single digit numbers, however as soon as I tried to add numbers in the tens and hundreds, or numbers with decimals I encountered an error.

To fix this I had to think of a different way of splitting up the string before I rearranged it into postfix format. For this I decided to use the Split() function, using this function by specifying a character that would be a separator the string would be broken into multiple smaller strings split by the assigned separator.

This meant that I had to modify what I had already written to work with strings instead of characters. While I was making these modifications I also added some" else if" statements to my postfix formatting method to allow for operations in parentheses to be prioritized.

```
//method to rearrange the Split() string from a infix format to a postfix format for easier calculation using a stack
1 reference
static List<string> ConvertToPostfix(string[] strArray)
{
    //create new empty array to hold rearranged results
    List<string> strlist = new List<string>();

    // initializing empty stack
    Stack<string> stack = new Stack<string>();

    //go through the string array
    //for each string
    foreach (String str in strArray)
    {
        //check the first character
        char c = str[0];
        //if the character is a digit or a negative add the string to the return list
        if (Char.IsDigit(c) || (c == '-' && str.Length > 1))
        {
            strlist.Add(str);
        }

        //if the first character is an '(', push it to the stack.
        else if (c == '(')
        {
            stack.Push(str);
        }
        //if the first character is an ')', pop and output from the stack until an '(' is encountered.
        else if (c == ')')
        {
            while (stack.Count > 0 && stack.Peek() != "(")
            {
                strlist.Add(stack.Pop());
            }
            if (stack.Count > 0 && stack.Peek() != "(")
            {
                // invalid expression
                return null;
            }
            else
            {
                stack.Pop();
            }
        }

        //if the first character is an operator
        else
        {
            //while there is something on the stack if the first(checking) characters priority is less
            //than the priority of the operator on the top of the stack
            while (stack.Count > 0 && OperatorPriority(c) <= OperatorPriority(stack.Peek()[0]))
            {
                //add the top of the stack to the list
                strlist.Add(stack.Pop());
            }
            //otherwise add the operator to the stack to check against next time a operator comes up
            stack.Push(str);
        }
    }

    // after all of the strings have been checked pop all the the operators from the stack to the string list
    while (stack.Count > 0)
    {
        strlist.Add(stack.Pop());
    }
    //return the string list
    return strlist;
}
```

This also meant that I had to alter the calculation method to work with a list of strings rather than characters as well.

```
//method to calculate the answer to the equasion using the postfix formatted string list
1 reference
static double CalculateAnswer(List<string> postfixExp)
{
    // create new temporary stack
    Stack<double> tempStack2 = new Stack<double>();

    //for each of the strings
    foreach (String str in postfixExp)
    {
        //if its a number either positive or negative
        char c = str[0];
        if (Char.IsDigit(c) ||(c == '-' && str.Length > 1))
        {
            //convert it into a double and add to the stack
            var n = Convert.ToDouble(str);
            tempStack2.Push(n);
        }

        //if the scanned string is an operator, pop two elements from stack apply the operator then push back
        else
        {
            double n1 = tempStack2.Pop();
            double n2 = tempStack2.Pop();

            switch (str)
            {
                case "+":
                    tempStack2.Push(n2 + n1);
                    break;

                case "-":
                    tempStack2.Push(n2 - n1);
                    break;

                case "/":
                    tempStack2.Push(n2 / n1);
                    break;

                case "*":
                    tempStack2.Push(n2 * n1);
                    break;

                case "^":
                    tempStack2.Push(Math.Pow(n2, n1));
                    break;
            }
        }
    }
    return tempStack2.Pop();
}
```

The final thing I did was write one final method, this was a formatting method that would format the initial string in a way that made splitting the string up with the Split() method easier. While also removing any other characters other than the operators and operands, in an attempt to reduce errors.

```
//method to format the string so that it can be easily be Split()
1 reference
static string FormatString(string str)
{
    //create empty string for return
    string outputString = string.Empty;

    //create an array of accepted characters to check the expression for
    char[] operatorArray = new char[8] { '^', '*', '/', '+', '-', '(', ')', '.' };

    //for each character in the string
    for (int i = 0; i < str.Length; i++)
    {
        //if the character is a number or a space add it to the formatted string
        if (Char.IsDigit(str[i]) || str[i] == ' ')
        {
            outputString += str[i];
        }
        //if its not a number
        else
        {
            //go through the operator array to see if the character matches one of the operators before adding it to the operator list
            //for error prevention this method should ignore anything that isnt an operator eg letters
            for (int j = 0; j < operatorArray.Length; j++)
            {
                //if it is an operator or decimal add it to the string
                if (str[i] == operatorArray[j])
                {
                    //if there isnt a gap between the perentheses and the next char add one to make splitting easier
                    if (str[i] == '(' && str[i + 1] != ' ')
                    {
                        outputString += str[i];
                        outputString += ' ';
                    }
                    else if (str[i] == ')')
                    {
                        outputString += ' ';
                        outputString += str[i];
                    }
                    else
                    {
                        outputString += str[i];
                    }
                }
            }
        }

    }
    //return the formatted string
    return outputString;
}
```

Question 4

For question 4 I decided to add factorials, because it seemed interesting and it was more of an opportunity to show off some error handling.

By this point the large majority of the code was written and I just needed to change a couple of things and write a method that would factorialize a number.

- Adding the factorial “!” to the allowed operators in my formatting method

```
//create an array of accepted characters to check the expression for
char[] operatorArray = new char[9] { '^', '*', '/', '+', '-', '(', ')', '.','!' };

//for each character in the string
for (int i = 0; i < str.Length; i++)
{
    //if the character is a number or a space add it to the formatted string
    if (Char.IsDigit(str[i]) || str[i] == ' ')
    {
```

- Make sure it split correctly by adding a space after it

```
//if there isnt a gap between the perentheses and the next char add one to make splitting easier
if (str[i] == '(' && str[i + 1] != ' ')
{
    outputString += str[i];
    outputString += ' ';
}
else if (str[i] == ')' ||  str[i] == '!'  && str[i - 1] != ' ')
{
    outputString += ' ';
    outputString += str[i];
}
else
{
    outputString += str[i];
}
```

- Add the factorial operator to the Operator Priority Method at the highest priority

```
//method to assign int priorities to the different calculations that will be used
2 references
static int OperatorPriority(char op)
{
    switch (op)
    {
        //addition and subtraction have the lowest priority
        case '+':
        case '-':
            return 1;

            //then multiplication and division have the second highest priority
        case '*':
        case '/':
            return 2;

            //powers
        case '^':
            return 3;
            //factorials
        case '!':
            return 4;

    }
    return -1;
}
```

- Add a separate else if statement in the calculate answer method that calls the factorial method, as unlike the other operators it only takes one operand. This could be expanded further later into its own switch case with other operators that only take one operand for example square root etc.

```
//for each of the strings
foreach (String str in postfixExp)
{
    //if its a number either positive or negative
    char c = str[0];
    if (Char.IsDigit(c) ||(c == '-' && str.Length > 1))
    {
        //convert it into a double and add to the stack
        var n = Convert.ToDouble(str);
        tempStack2.Push(n);
    }

    //if the scanned string is an operator that only takes one operand (factorial) calculate
    else if(str == "!")
    {
        double num1 = tempStack2.Pop();
        tempStack2.Push(CalculateFactorial(num1));
    }

    //if the scanned string is an operator, pop two elements from stack apply the operator then push back
    else
    {
        double n1 = tempStack2.Pop();
        double n2 = tempStack2.Pop();

        switch (str)
        {
            case "+":
                tempStack2.Push(n2 + n1);
                break;

            case "-":
                tempStack2.Push(n2 - n1);
```

- Finally, the calculate factorial method, calculates the factorial and contains the error code to stop the computer crashing from trying to factorialize a negative number

```
//method to calculate the factorial of a number
1 reference
static double CalculateFactorial(double inputNum)
{
    //added error handling in the factorial method to make sure that the program doesnt crash
    if (inputNum < 0)
    {
        Console.Error.WriteLine("cant factorialise a negative number");
        return inputNum;
    }
    else
    {
        //factorialise the number
        double factorialNum = inputNum;
        for (double i = factorialNum - 1; i > 0; i--)
        {
            factorialNum *= i;
        }
        return factorialNum;
    }
}
```