**Calculator  
Learn about Reverse Polish Notation by building a simple calculator**

TEACHER’S GUIDE

Created by Richard Pawson

[Creative Commons License](https://creativecommons.org/licenses/by-sa/4.0/)  
This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

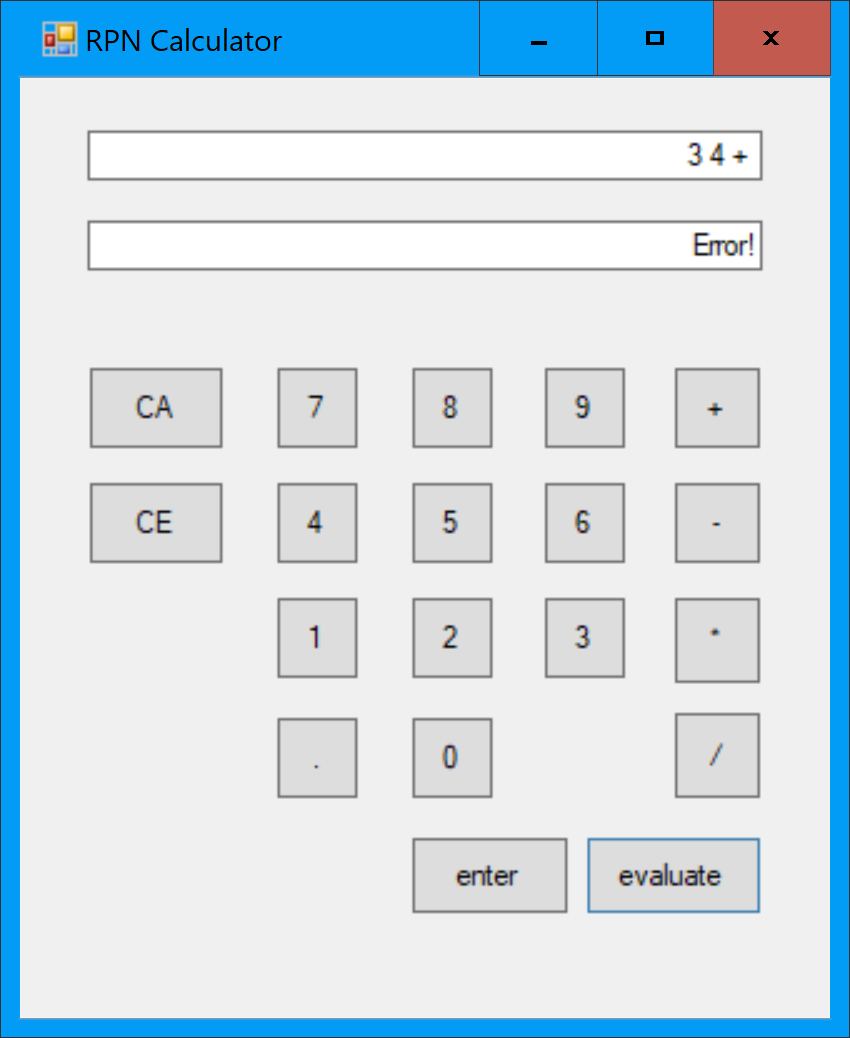
Contents

[Model answers 3](#_Toc508639845)

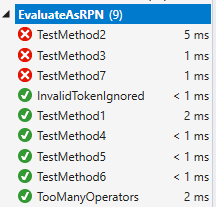
[Complete code for Core.cs 7](#_Toc508639846)

# Model answers

1. Paste in a partial screenshot showing the whole calculator after entering the expression above and hitting evaluate.



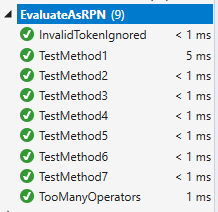
1. Paste in a partial screenshot showing which of the EvaluateAsRPN tests pass, and which fail.



1. Double click on *each test* to examine the test scenario in code. Which operation(s) are causing the failures?

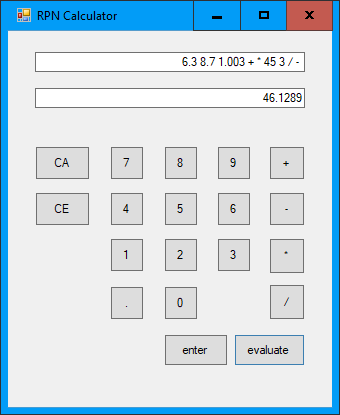
It is failing on the minus and divide operations.

1. Verify that all the EvaluateAsRPN tests pass (the other sets of tests will still fail for now) and paste in a partial screenshot showing this.



1. Paste in a screenshot showing the calculator displaying the full expression and the result.

For example:



1. Show your working by sketching a stack and showing the values on the stack at each step. .

Token processed:

7 6 3 4 + \* -

Stack after each step:

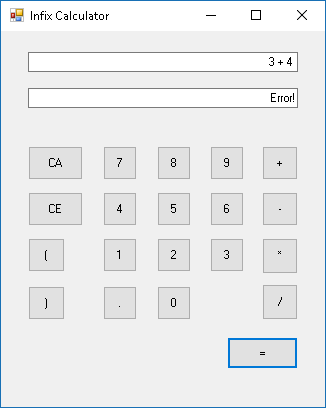
7 6 3 4 7 42 -35 (answer)

7 6 3 6 7

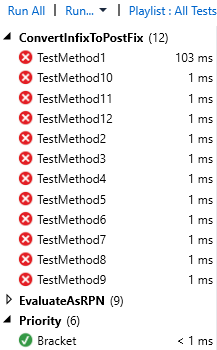
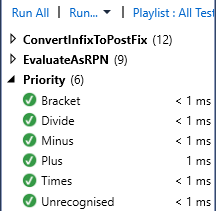
7 6 7

7

1. Paste in a partial screenshot showing the error.



1. Paste in a partial screenshot showing passing and failing tests.



1. Trace though this algorithm on paper using the specific expression shown in the diagram above ( A+B\*C\_D ). What is the order of tokens in the output once the algorithm has been completed?
2. Input: A+B\*C-D

Output:

Siding:

1. Input: +B\*C-D

Output: A

Siding:

1. Input: B\*C-D

Output: A

Siding: +

1. Input: \*C-D

Output: A B

Siding: +

1. Input: C-D

Output: A B

Siding: + \*

1. Input: -D

Output: A B C

Siding: + \*

1. Input: D

Output: A B C \* +

Siding: -

1. Input:

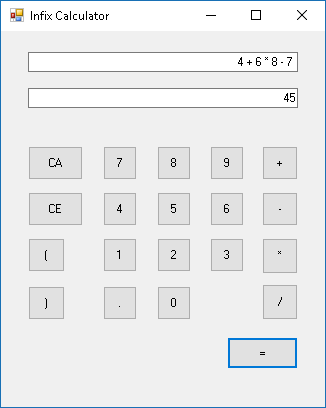
Output: A B C \* + D -

Siding:

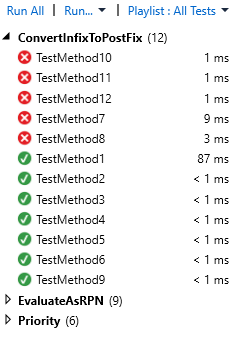
1. In tracing through the algorithm for that same example, what is the maximum number of operators that are being held on the stack at any point?

2

1. Paste in a screenshot showing the expression and the result on the calculator



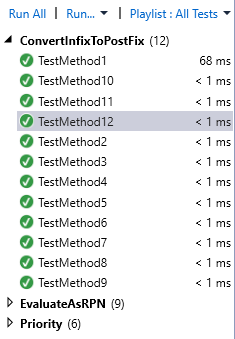
1. Paste in a screenshot showing which tests pass and fail.



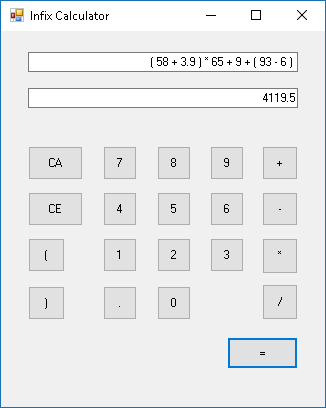
1. Looking at the code for the tests that fail, what do they have in common?

They all contain brackets.

1. Paste in a screenshot showing all tests passing.



1. Paste in a screenshot of the calculator showing both the expression and the result



# Complete code for Core.cs

using System;

using System.Collections.Generic;

using System.Text;

namespace Calculator

{

public class Core

{

private List<object> Tokens = new List<object>();

public void Clear()

{

Tokens = new List<object>();

}

internal void AddSymbolAsToken(char symbol)

{

Tokens.Add(symbol);

}

public double AddNumberAsToken(string numberAsText)

{

double number = Convert.ToDouble(numberAsText);

Tokens.Add(number);

return number;

}

public string TokensAsString()

{

var sb = new StringBuilder();

foreach (var token in Tokens)

{

sb.Append(token.ToString()).Append(" ");

}

return sb.ToString();

}

public double EvaluateTokensAsRPN()

{

return EvaluateAsRPN(Tokens);

}

public static double EvaluateAsRPN(List<object> Tokens)

{

double result = 0;

var stack = new Stack<double>();

foreach (object token in Tokens)

{

if (token is double)

{

stack.Push((double)token);

}

else

{

switch ((char)token)

{

case '+':

stack.Push(stack.Pop() + stack.Pop());

break;

case '-':

var b = stack.Pop();

var a = stack.Pop();

stack.Push(a - b);

break;

case '\*':

stack.Push(stack.Pop() \* stack.Pop());

break;

case '/':

var d = stack.Pop();

var c = stack.Pop();

stack.Push(c / d);

break;

}

}

}

result = stack.Pop();

return result;

}

public double EvaluateTokensAsInfix()

{

var tokensAsRPN = ConvertInfixToPostfix(Tokens);

return EvaluateAsRPN(tokensAsRPN);

}

public static List<object> ConvertInfixToPostfix(List<object> inputTokens)

{

var s = new Stack<char>();

var outputList = new List<object>();

foreach (var t in inputTokens)

{

if (t is double) //token is a value

{

outputList.Add(t); //send it straight to the output

}

else

{

char token = (char)t; ///... so cast it to a char

if (token == '(')

{

s.Push(token);

}

else if (token == ')')

{

while (s.Count != 0 && !s.Peek().Equals('('))

{

outputList.Add(s.Pop());

}

s.Pop();

}

else

{

while (s.Count != 0 && Priority(s.Peek()) >= Priority(token))

{

outputList.Add(s.Pop());

}

s.Push(token);

}

}

}

while (s.Count != 0) //Unload any remaining operators onto the stack

{

outputList.Add(s.Pop());

}

return outputList;

}

public static int Priority(char c)

{

if (c == '\*' || c == '/')

{

return 2;

}

else if (c == '+' || c == '-')

{

return 1;

}

else return 0;

}

}

}