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

STUDENT WORKBOOK

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/).

[Introduction 3](#_Toc508640201)

[Exercise 1. Exploring the skeleton solution 5](#_Toc508640202)

[Exercise 2. Implement the RPN Calculator 9](#_Toc508640203)

[Exercise 3. Implement the Infix Calculator 13](#_Toc508640204)

[The Shunting Yard Algorithm 14](#_Toc508640205)

[Exercise 4. Adding brackets 17](#_Toc508640206)

## Introduction

When we write down the following expression

a \* (b + c) / d

we are actually using what is known ‘infix notation’ - which means that the operators (\*,+,/) are located **in** between the two operators that they are applied to. This might seem like the obvious, even natural, way to write the expression, but in truth that is only because it is the one we learned from a young age - so it is the most familiar to us. But there are two other ways to write the same expression. This next one:

/ \* a + b c d

is known as ‘prefix notation’ because in each case the operator **pre**cedes the two operands it applies to. Notice that there are no brackets. This might seem a very strange way to write the expression. But if you were used to coding in a ‘Functional Programming Language’ (such as Haskell) it would be at least somewhat more familiar to you. It is sometimes referred to as ‘Polish Notation’, after the nationality of the logician, [Jan Łukasiewicz](https://en.wikipedia.org/wiki/Jan_%C5%81ukasiewicz), who invented it in 1924.

Our third variant:

a b c + \* d /

is known as ‘postfix’ notation because in each case the operator comes **post**(after) the operands it applies to. This notation is also known as ‘*Reverse* Polish Notation’ (or just RPN).

RPN is important in computer science because, surprisingly perhaps, it is the easiest form of an expression for a computer to interpret and hence calculate.

For this reason, the first electronic pocket calculators used RPN. One of the most famous examples was the Hewlett-Packard HP-35 pictured below:



Introduced in 1972, this was the first pocket calculator to have scientific functions, and was also the first pocket calculator to be taken into space. Notice that it has no ‘=’ key. To calculate the sum of 3 and 4 you would have pressed the keys: 3 [Enter] 4 +. That might seem awkward, but it suited the calculator - and it made it possible to calculate expressions such as (3+4) \* (5+6), without the need for brackets by typing: 3 [Enter] 4 + [Enter] 5 [Enter] 6 + \*.

Over time, however, as the speed and memory capacity of silicon chips improved, pocket calculators reverted to the more familiar infix notation.

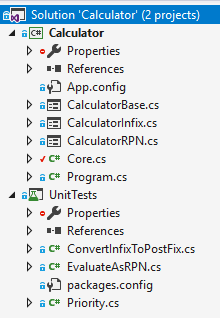
To evaluate an expression such as (3 + 4) \* (5 + 6), these early calculators introduced a memory key (M) to store and retrieve intermediate results. Later they introduce bracket keys. Later still calculators were given multi-line displays so that you could see, even edit, the whole expression on one line, then display the result on another line.

In this worksheet we are going to create a simulation of a two basic calculators: first one that uses RPN, and then a more conventional modern calculator using infix notation. We will create the RPN one first, because the infix calculator will actually convert the infix notation to RPN and then delegate the calculation to the same function that evaluates RPN.

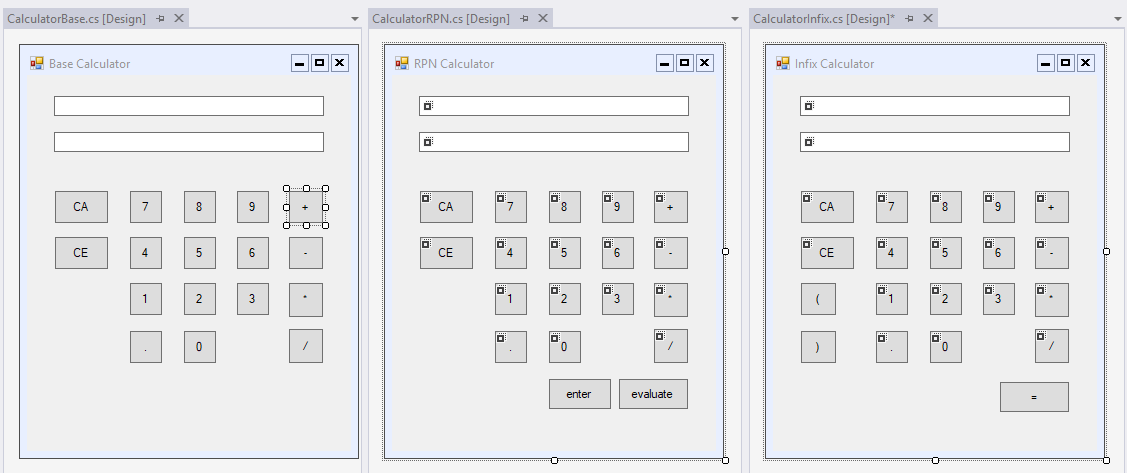
# Exploring the skeleton solution

Download the Calculator - Skeleton.zip file and *unzip it into a suitable working directory.* Then open the Calculator.sln solution file using Visual Studio.

The Calculator solution is a ‘skeleton’ - it has the bones or structure in place to hold the code together, but you will need to ‘flesh it out’ to get it working. It consists of two projects, **Calculator** and **UnitTests** as shown below in the **Solultion Explorer** pane:



We’ll come back to UnitTests later; for the moment just explore the Calculator project. It uses the Windows Forms library to create graphical user interfaces, and the project contains three forms: CalculatorBase, CalculatorInfix, and CalculatorRPN. In the screenshot below, the three forms are displayed side by side, in their **[Design]** view, for comparison:



You can see that CalculatorBase defines the two displays and all the keys that are common to both the other two calculators. CalculatorRPN adds the **Enter** and **Evaluate** keys, while CalculatorInfix adds the **=**  and brackets **( )** keys. From the solution explorer we can *right-click*  on one of the form files and select *View Code*, to see the C# code that corresponds to each GUI, for example:

public partial class CalculatorRPN : CalculatorBase

{

public CalculatorRPN() : base()

{

InitializeComponent();

}

private void enter\_Click(object sender, EventArgs e)

{

TransferNumberToExpression();

}

private void evaluate\_Click(object sender, EventArgs e)

{

try

{

DisplayResult(Calculator.EvaluateTokensAsRPN());

}

catch (Exception)

{

numericDisplay.Text = "Error!";

}

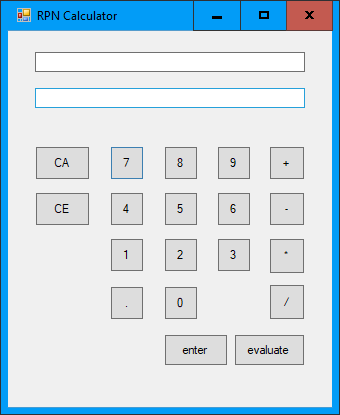
}

}

If you are already familiar with object-oriented programming (OOP) you will recognise in the code above that CalculatorRPN *inherits* from CalculatorBase, which makes sense. If you are not already familiar with OOP don’t worry - you do not need to understand OOP to continue with this worksheet.

The two functions enter\_Click and evaluate\_Click are called automatically when the program is running and the user clicks on the **enter** or **evaluate** key respectively. (You can verify this by putting a breakpoint within those functions.) CalculateBase defines equivalent functions for all of the common keys it provides.

Look at the **Program.cs** file and notice that it is set up to create a CalculatorRPN. Run the project, and notice that the window that pop’s up is indeed of this type:



Using the mouse, key in the following expression:

**3 [enter] 4 + 5 \***

The top display holds the expression, the lower display holds numbers while they are being entered, and will (eventually) display the result of evaluating the expression.

**CA** (Clear All) will clear the whole expression; **CE** (Clear Entry) clears only the numerical (lower) display.

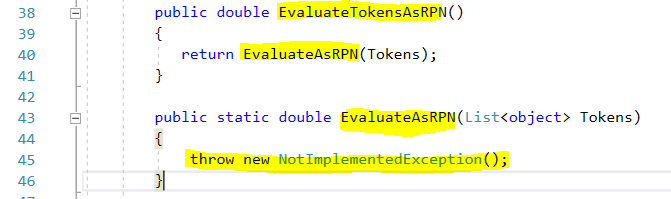
To calculate the result hit the **evaluate** key. You will get an error!

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

Going back to the code above we can see, that hitting evaluate would cause this line to execute:

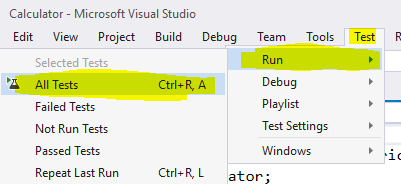
DisplayResult(Calculator.EvaluateTokensAsRPN());

If you mouse over the EvaluateTokensAsRPN part, *right-click* and select **Go To Definition**, you will see (in the Core.cs file) the reason for the error:

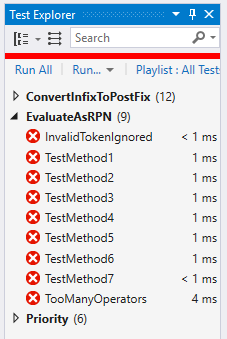


The underlying EvaluateAsRPN function hasn’t been implemented! That’s your job. You’ll need to delete the line throw new NotImplementedException(); and replace it with a proper implementation.

Before we tackle that, let’s just take a quick look at the unit tests. You can run all the tests simply by selecting Test > Run > All Tests:



And you should see the Test Explorer window open, and the result that *all* the tests have failed:



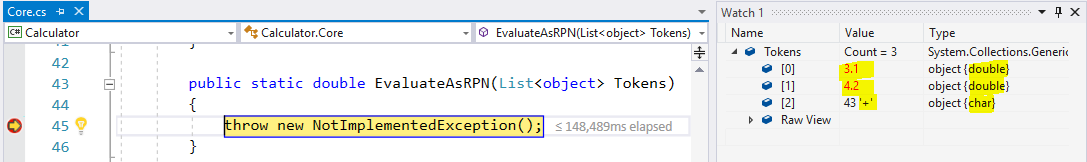
For the screenshot above, the ConvertInfixToPostfix tests and Priority tests have been collapsed to save space, and because the first tests we want to get passing are for the function EvaluateAsRPN. Once you have all of *those* tests passing, your RPN calculator should be fully functional. The other two sets of tests become relevant when we implement the Infix Calculator later on - it is OK that they are failing for now.

# Implement the RPN Calculator

To get the RPN Calculator working, we just need to implement the function EvaluateAsRPN, found in the Core.cs file. Put a breakpoint in the function, run the program with this expression:

**3.1 [enter] 4.2 + [evaluate]**

and inspect the tokens parameter (using *right-click* **Add Watch** on it) you would see that the tokens list contains three elements, which you can expand, thus:



Don’t worry if you don’t understand *everything* that you are seeing here - a general idea will suffice. The first two elements are of type double and represent the numbers 3.1 and 4.2 respectively. The third element is of type char and represents the ‘+’ key (the ASCII value for ‘+’ is 43).

We need to work through each of the elements in the list and then handle the double and char types separately, so we can create a ‘skeleton’ for our function like this (the comments are for clarity - you don’t need to copy them, but make sure you understand the structure of the skeleton code):

public static double EvaluateAsRPN(List<object> Tokens)

{

double result = 0;

foreach (object token in Tokens)

{

if (token is double)

{

//TODO: handle the numbers

}

else //Assume it is a char

{

switch ((char)token) // ‘(char)’ is to ‘cast’ the token into a char

{

case '+':

//TODO: handle the plus operation, etc.

break;

case '-':

//TODO

break;

case '\*':

//TODO

break;

case '/':

//TODO

break;

}

}

}

return result;

}

Now we come to the key idea in evaluating an RPN expression: the use of a *stack*.

You should already have learned a little about stacks as a simple form of *data structure*. You should be aware that it is a LIFO (Last In, First Out) mechanism. It operates like a stack of plates: you can put a plate on top of the stack (the operation is known as *push)* and remove the top plate only (known as *pop)*. You can also *peek* to inspect the topmost item without removing it. You probably know that you can implement a stack in code by using an array (or a list) and maintaining a pointer to the top-most item at any stage. But we are going to use a ready-made stack that comes as part of the standard .NET libraries.

An RPN expression can be evaluated, reading the expression left to right, one *token* at a time, using the following algorithm:

* If the token is a value, push it onto the stack.
* If the token is an operator, pop the top two items from the stack (if you have implemented the algorithm correctly they will always be values), perform the operation on them, and push the result back onto the stack.
* When you get to the end of the tokens, if it is a valid expression there will be just one item left on the stack - a value representing the result. Pop the item and return it.

You can implement this function with these code changes:

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 '-':

stack.Push(stack.Pop() - stack.Pop());

break;

case '\*':

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

break;

case '/':

stack.Push(stack.Pop() / stack.Pop());

break;

}

}

}

result = stack.Pop();

return result;

}

Make the changes and check that you understand the general idea of the code. Then run the tests. Some of the tests under EvaluateAsRPN will now pass, but some will still fail.

1. Paste in a partial screenshot showing which of the EvaluateAsRPN tests pass, and which fail.
2. Double click on *each test* to examine the test scenario in code. Which operation(s) are causing the failures?

The problem is that the ‘minus’ and ‘divide’ operators are not *commutative* - the order of the operands makes a difference - unlike ‘plus’ and ‘times’, which are commutative. Our implementation is interpreting the tokens ‘3 4 –‘ as 4 minus 3, when we want it to mean 3 minus four. Similarly for divide. So make these changes:

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;

}

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.

Now you can run the Calculator itself. Try out a fairly complex (but sensible) expression and

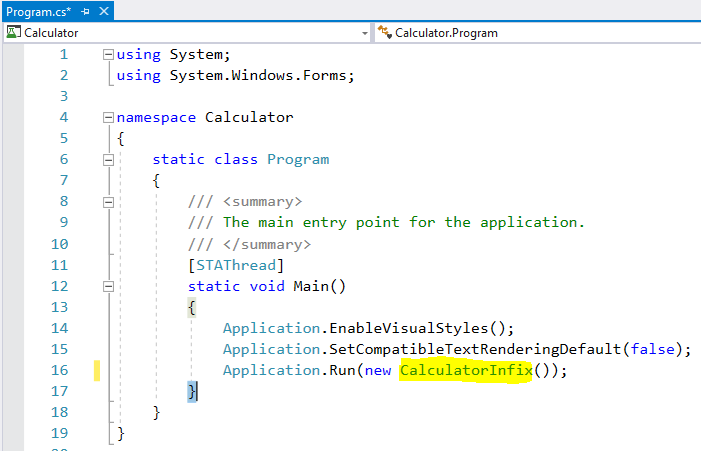
1. Paste in a screenshot showing the calculator displaying the full expression and the result.

Finally, while we’re still thinking about RPN, go back to the explanation of the algorithm a couple of pages back. Use that algorithm manually (without the aid of the calculator) to evaluate the following RPN expression: 7 6 3 4 + \* -

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

# Implement the Infix Calculator

We can now turn our attention to the Infix calculator. Start by changing the code in the **Program** file, so that it uses the CalculatorInfix form:



Run the program, try out a simple calculation (such as **3 + 4 =** ) and show that you get an error, as you did at the start of the previous exercise.

1. Paste in a partial screenshot showing the error.

If you were to trace through the code (as we did before) from the equals\_Click function in the CalculatorInfix code, you will end up at this code:

public double EvaluateTokensAsInfix()

{

var tokensAsRPN = ConvertInfixToPostfix(Tokens);

return EvaluateAsRPN(tokensAsRPN);

}

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

{

throw new NotImplementedException();

}

public static int Priority(char c)

{

throw new NotImplementedException();

}

The EvaluateTokensAsInfix function actually delegates to the EvaluateAsRPN function (which we know now works), but first calling another function: ConvertInfixToPostfix - and that function hasn’t been implemented yet.

There’s also another function that hasn’t been implemented yet: Priority. This isn’t yet called from anywhere, but we are going to need it to implement the one above.

Unlike processing RPN, when you evaluate an Infix expression, you need to understand the rules of operator priority (or *precedence)* – which you will have learned in mathematics (remember ‘BIDMAS’ ?). We’ll start with that, because it is easy to write and because we have separate unit tests for it. For any allowed operator (as a char) we need to get back a number representing its relative priority: 2 for multiply and divide; 1 for plus and minus; 0 for anything else. (If our calculator had a ‘raise to the power of’ function then that would be priority 3).

public static int Priority(char c)

{

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

{

return 2;

}

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

{

return 1;

}

else return 0;

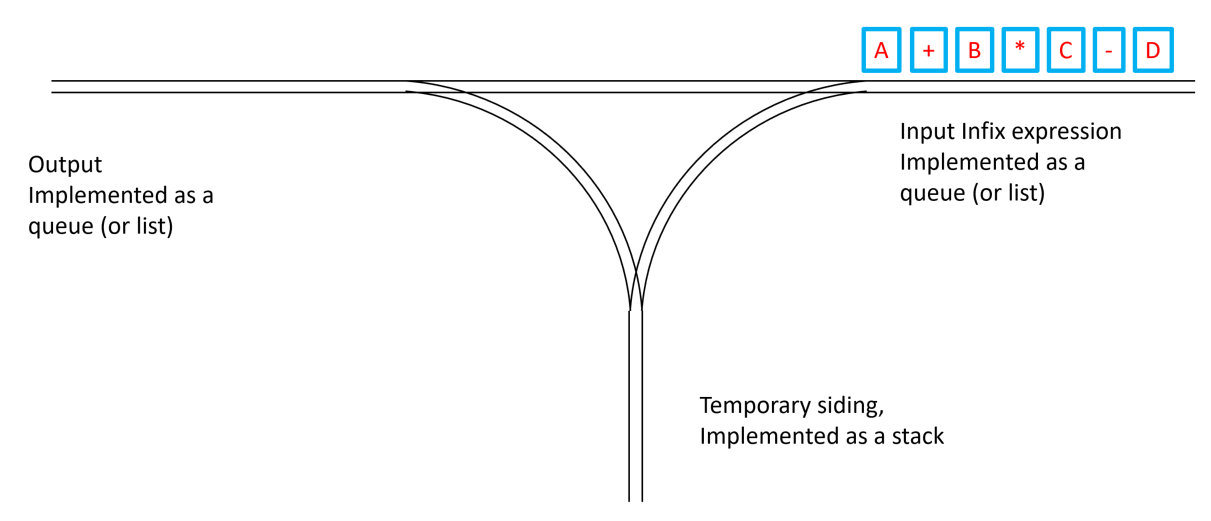
}

Make the code changes above then run all the tests and show that the *Priority* tests are now all passing. Only the ConvertInfixToPostFix tests should now be failing.

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

## The Shunting Yard Algorithm

An effective algorithm for converting an infix expression to postfix, making use of operator priority, is known as the Shunting Yard Algorithm. It was devised by Computer Scientist Edsger Dijkstra, who also devised the more famous algorithm for finding the shortest path through a graph). The Shunting Yard Algorithm can be envisaged as railway layout consisting of a straight-through track and a siding. The input expression is represented as a queue (or just a list) of ‘trucks’, one for each token. The output line is also implemented as a queue or list. The siding is implemented as a stack, in order to achieve Last-In First-Out behaviour.



Tokens are moved off the front of the input queue/list one at a time, following these rules:

* If the token is a value (a literal number, or a variable or constant representing a number) then it is moved straight across to the output queue.
* If the token is an operator then it is pushed *towards* the siding. However …
* If the stack already contains an operator that has priority (BIDMAS) equal to or higher than the one about to be pushed, then that operator is ‘popped’ from the stack and sent to the output queue - the process repeated until the stack is empty OR the topmost operator in the stack is *lower* priority than the incoming one. Then the new operator is pushed onto the stack.
* When the input queue is emptied, then any remaining operators on the stack are unloaded onto the output queue.

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. 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?

The following is a basic implementation of the ConvertInfixToPostFix method:

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 //Is is an operator...

{

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

//Then check its priority against any operators held in the stack

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;

}

Add the code, making sure that you follow what it is doing, and then run the program. Try the calculator with the expression as shown above.

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

Now run the tests again. Some of the ConvertInfixToPostFix tests should now pass, some fail.

1. Paste in a screenshot showing which tests pass and fail.
2. Looking at the code for the tests that fail, what do they have in common?

# Adding brackets

To get the brackets working, the basic algorithm needs these new rules:

* If the token is an open-bracket, push it onto the stack
* If the token is a close-bracket, unload tokens from the stack to the output *until you reach the first open-bracket,* then discard (destroy) both the open and close brackets. (OK, so this is where the metaphor of a shunting yard isn’t perfect!)

Here are the necessary code changes:

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;

}

Make the changes and run the tests.

1. Paste in a screenshot showing all tests passing.

Run the Infix calculator and try out an expression of your own that involves at least two sets of brackets.

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