Calculator Application Design Document

Brandon Lyman | January 2022

# Overview

This application will consist of a calculator capable of performing primitive operations on numeric values. The calculator will be represented in 2-Dimensional space using buttons and a text display. The user will be able to select integers 0 through 9 as values to operate upon. They may also create decimal values by using a decimal point. They will be able to perform further operations on values resulting from the primitive operations. Finally, while the user is building their expression to evaluate, they will be able to clear their last entry (backspace), or the entire expression. The calculator will support an arbitrary number of custom operations that can be selected from a drop-down menu by the user. When a custom operation is selected, a dialog box will appear allowing the user to enter all necessary arguments to the operation, and the result will appear on the calculator’s display. The calculator app will be shipped with three custom operations: the Pythagorean solve operation, factorial operation, and an inches-to-centimeters converter operation.

This application will be programmed in the Unity/C# environment. It will be shipped on PC/Mac/Linux systems and Android systems. It is for educational purposes only and will not be marketed as a commercial product.

# Application Design

User input will be stored in infix notation as a string array. The application will accept all rational numbers as operands. Upon confirmation by the user (selecting the “=” button or equivalent), the infix expression will be converted to a postfix expression. The shunting-yard algorithm will be used to execute this conversion, imposing operator precedence and associativity order as the resulting postfix expression is built. Once the final postfix expression is constructed, a stack will be used to evaluate the it to obtain the result of the user’s initial infix expression.

## Operator Precedence and Associativity

The order of operations are as follows, in order of highest to lowest operator precedence[[1]](#footnote-1):

1. Exponentiation and root extraction
2. Multiplication and division
3. Addition and subtraction

A precedence value will be assigned to operators based on this hierarchy to ensure the application properly applies the order of operations. The precedence value will be stored in a dictionary, using the operator represented as a string for the dictionary’s key and the precedence value as the dictionary’s value. Parenthesis (parens) will be assigned the lowest precedence value among operators. This precedence value will be referenced during the infix-to-postfix expression conversion. Note that root extraction will not be included as a primitive operation in this application.

Operators with the same precedence will be evaluated based on their associativity behavior. The primitive operations for multiplication, division, addition, and subtraction are left-associative, and the exponentiation operation is right-associative. Associativity helps determine in which order two operations of the same precedence should be evaluated. If two operators have the same precedence and the incoming operator is left-associative, the incoming operator will be treated during the conversion as if it has a lower precedence value than the other operator. Similarly, if two operators have the same precedence and the incoming operator is right-associative, the incoming operator will be treated during the conversion as if it has a higher precedence value than the other operator.

## Infix-to-Postfix Expression Conversion

The purpose of this step is to convert an expression that is more readable to humans (infix) to one that is more readable by the program (postfix). The algorithm to convert an infix expression to a postfix expression is derived from the shunting-yard algorithm and is as follows[[2]](#footnote-2):

1. Create an empty stack for operator storage (referred to hereafter as “S”), and an empty string to store the result of the conversion (referred to hereafter as “R”).
2. Parse the infix expression array from left to right. For each token:
   1. If the token is an operand, append it to R.
   2. If the token is a left paren, push it into S.
   3. If the token is a right paren, pop S until a left paren is popped. Append each operator to R, excluding the parenthesis.
   4. If the token is an operator (\*,/,+,-,^), push it into S, but first pop any existing operators in the stack with a higher precedence (or an equal precedence if the token operator is left-associative) and append them to R.
3. When the infix expression is fully parsed, pop each of the remaining operators in S and append them to R.

The resulting postfix expression can be evaluated to obtain the resulting value of the original infix expression.

## Postfix Expression Evaluation

The algorithm to evaluate the postfix expression read in as a string array are as follows[[3]](#footnote-3):

1. Create an empty stack for operand storage (referred to hereafter as “S”).
2. Parse the postfix expression array from left to right. For each token:
   1. If the token is an operand, convert it to a float and push the value into S.
   2. If the token is an operator, pop the operand stack twice. The first pop will return the right operand, and the second pop will return the left operand. Perform the primitive operation, and push the resulting value into S.
3. When the postfix expression is fully parsed, the result is at the top of the stack. Pop the operand stack to return the value.

A switch statement will be used to determine which primitive operation should be performed. The application is concerned with only 5 primitive operations, and additional primitive operations are not anticipated at this time. Therefore, it is logical, concise, and intuitive to use a switch statement.

The result of the postfix expression evaluation will be displayed on the frontend calculator display and stored as a current value such that the user may continue to operate on the resulting value.

## Custom Operations

The calculator application will support an arbitrary number of custom operations. The architecture described in this section allows designers to add custom operations to the calculator application at their discretion, so long as they follow certain guidelines for creating a custom operation.

The crux of this architecture is Unity’s ScriptableObject class coupled with the strategy pattern. The strategy pattern software design pattern that enables selecting an algorithm at runtime - Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use[[4]](#footnote-4). In Unity, each custom operation will be represented by a ScriptableObject that implements an interface (ICustomOperation). This interface will ensure that the custom operation object has an “Execute” method that takes in a string array representing arguments to the operation and returns a string representing the result of the custom operation. The calculator application will then hold a reference to a list of ICustomOperations that can be selected from a dropdown list by the user. Once the user selects a custom operation, a dialog window will appear presenting the user with input fields representing each of the necessary arguments to the custom operation. The user can then execute the custom operation and the result will be stored and displayed as the current value of the calculator.

The calculator application will be shipped with three custom operations to demonstrate this behavior: The Pythagorean Solve operation, the factorial operation, and the inches-to-centimeters converter operation.

### Pythagorean Solve Operation

The Pythagorean theorem is a method of determining the length of one side of a triangle given the length of the other two sides of the triangle. The Pythagorean solve custom operation will prompt the user to enter side lengths for a right triangle, expecting the user to leave one side empty (or set it equal to 0). The operation will then use the Pythagorean theorem to calculate the missing side of the triangle and return it as the operation’s result.

### Factorial Operation

The factorial operation (!) returns the product of all positive integers less than or equal to the operand[[5]](#footnote-5). The factorial custom operation will prompt the user to enter a positive integer. A simple recursive algorithm will be used to calculate the value of the factorial. The operation will then calculate the factorial of the given operand and return it as the operations result.

### Metric Converter Operation (Inches-to-Centimeters)

A length represented in inches can be converted to centimeters by multiplying the operand by 2.54. The metric converter custom operation will prompt the user to enter a positive number representing a length. The operation will then convert the length to centimeters and return it as the operation’s result.

## Error Handling

The user input will be restricted by the buttons present on the frontend calculator visual. For the most part, this will eliminate the user’s ability to input invalid tokens such as letters and symbols that are not primitive operators. The frontend design does still allow for a few erroneous inputs, and the associated error handling methodology will be described here.

### Mismatched Parentheses

It is possible for the user to enter an infix expression with mismatched parentheses. Parentheses are considered mismatched if there is an unequal amount of left and right parenthesis. This error will be handled when the user’s infix expression is converted to a postfix expression. If the conversion algorithm encounters mismatched parenthesis, the following error message will be logged and relayed to the frontend if an input fails this test:

*“Syntax Error: Mismatched parenthesis in expression.”*

### Operands with Multiple Decimal Points

It is possible for the user to enter a decimal operand with multiple decimal points. Operands such as this have no mathematical meaning, and therefore should be considered an error. This error will be prevented while the user is entering an operand. While the operand is being constructed by the user, a string will store the user’s current operand input. If this string already contains a decimal point, another will not be added when the user selects the “.” button an additional time. The frontend display will reflect this current operand input string, implicitly signaling to the user that they are not allowed to enter a multi-decimal point number.

### Leading Zeros

It is possible for the user to enter an arbitrary number of zeros before entering a non-zero numerical value. A single leading zero is mathematically meaningful (albeit unnecessary in C#) if the number is a decimal that is less than 1 (ex. 0.5). Otherwise, leading zeros do not add information to the operand and are therefore unnecessary. Leading zeros will be prevented while the user is entering an operand. While the operand is being constructed by the user, a string will store the user’s current operand input. If a zero already exists in the current operand input string, and the string does not also contain a decimal point, another zero will not be added when the user selects the “0” button an additional time. The frontend display will reflect this current operand input string, implicitly signaling to the user that they are not allowed to enter leading zeros before a decimal point.

### Consecutive Operators

It is possible for the user to enter two operators consecutively without entering an operand between them. This has no mathematical meaning. Consecutive operators will be prevented while the user is entering the infix expression. While the operand is being constructed by the user, a string will store the user’s current infix expression input. If an operator is selected directly after another operator, the original operator will be replaced by the incoming operator. The frontend display will reflect this current operand input string, implicitly signaling to the user that they are not allowed to enter consecutive operators.

### Errors Resulting from Custom Operations

Recall the calculator will support an arbitrary number of custom operations. Given the wide variety of custom operations that can be created for the application, an appropriate error handling methodology must exist for each custom operation. Custom operations will be required to send error messages to the frontend display and to log the messages to the console, to ensure the user is aware of what has caused the error. The calculator display will read “Syntax Error”, and the message will be displayed on the calculator’s console display. The errors that could result from the custom operations shipped with this application are described here. Note that errors such as operands with multiple decimal points and leading zeros will be prevented in the same manner as described above.

It is possible for the user to enter a letter or symbol for each of these custom operations, as they are no longer restricted by the frontend calculator design when the dialog box is presented. Input of letters or symbols into input fields meant for numerical values will be prevented while the user is entering their input. While the input is being entered, each value will be validated to ensure it is either a number or a decimal point. Note that each of the three included custom operations do not allow negative numbers as inputs.

#### Pythagorean Solve

The Pythagorean theorem is a method of determining the length of one side of a triangle given the length of the other two sides of the triangle. Inputs for this operation must be greater than 0.

##### Invalid Lengths

Negative inputs, and inputs of length 0 are mathematically meaningless because this theorem deals with lengths. Negative inputs will be prevented by disallowing the user to enter a negative sign. An invalid lengths error will be tested when the user attempts to execute the operation. Before the result is calculated, each input will be tested to ensure it is greater than 0. The following error message will be logged and relayed to the frontend if an input fails this test:

*“Syntax Error: Invalid length detected. Please ensure all inputs are greater than 0.”*

##### Invalid Input Count

The Pythagorean Solve operation requires two inputs but has input space for three. This is because it is designed to allow the user to chose which side of the triangle they would like to solve for. Therefore, one of the three input fields must be left blank. If all three input fields are populated by the user, or fewer than two input fields are populated by the user, an error is thrown.

This error will be tested when the user attempts to execute the operation. Before the result is calculated, each input will be tested to ensure that exactly two inputs are populated. The following error message will be logged and relayed to the frontend if that test is failed:

*“Syntax Error: Incorrect number of inputs detected. Please ensure that two input fields are populated, and one remains blank.”*

##### Invalid Triangle

The Wikipedia page for the Pythagorean Theorem states “[a] corollary of the theorem is that in any right triangle, the hypotenuse is greater than any one of the other sides, but less than their sum[[6]](#footnote-6).” Therefore, if the operation is given the hypotenuse and a single leg of the triangle, the triangle would be invalid if the hypotenuse was shorter than the given side. This error will be referred to as an “invalid triangle,” or a shape that cannot be constructed from three sides and three angles given the two lengths.

This error will be tested when the user attempts to execute the operation. Before the result is calculated, if a leg of the triangle is given along with the triangle’s hypotenuse, the lengths will be compared. If the hypotenuse is shorter than the given leg, the following error message will be logged and relayed to the frontend:

*“Syntax Error: Invalid triangle detected. Please ensure given leg (a or b) is shorter than the given hypotenuse (c).”*

#### Factorial Operation

The factorial of a non-negative integer *n* is the product of all positive integers less than or equal to *n[[7]](#footnote-7)*. Inputs for this operation must be integers that are non-negative.

##### Null Input Error

This operation requires exactly one input. Before the result is calculated, the input will be tested to ensure it is not empty or null. The following message will be logged and relayed to the frontend if the input fails this test:

*“Syntax Error: No input detected. Please enter a number in the input field.”*

##### Non-integer Error

Inputs for this operation must be integers. A non-integer error will be tested when the user attempts to execute the operation. Before the result is calculated, each input will be tested to ensure it is an integer using C#’s native int.TryParse method. The following message will be logged and relayed to the frontend if an input fails this test:

*“Syntax Error: Non-integer input detected. Please ensure all inputs are integers.”*

##### Negative Error

Inputs for this operation must be non-negative. Negative inputs will be prevented by disallowing the user to enter a negative sign.

#### Inches-To-Centimeters Conversion

This operation is a simple length conversion. Like the Pythagorean Solve operation, inputs for this operation must be greater than 0 because the inputs are representing length.

##### Null Input Error

This operation requires exactly one input. Before the result is calculated, the input will be tested to ensure it is not empty or null. The following message will be logged and relayed to the frontend if the input fails this test:

*“Syntax Error: No input detected. Please enter a number in the input field.”*

##### Invalid Lengths

Negative inputs, and inputs of length 0 are mathematically meaningless because this conversion deals with lengths. Negative inputs will be prevented by disallowing the user to enter a negative sign. An invalid lengths error will be tested when the user attempts to execute the operation. Before the result is calculated, each input will be tested to ensure it is greater than 0. The following error message will be logged and relayed to the frontend if an input fails this test:

*“Syntax Error: Invalid length detected. Please ensure input is greater than 0.”*

## Prototyping

The application will be prototyped in the pure C# environment. A complete prototype will include a fully functional primitive operation calculator and logic for each of the three custom operations to be shipped with the application. The application prototype will be able to run from a static main method.

The prototype will consist of one interface (ICustomOperation) and five main classes: the Input Validator, the Pythagorean Solver, the Factorial Operator, the Metric (inches-to-centimeters) Converter, and the Calculator. Below are high level descriptions of each prototyped class and interface.

### ICustomOperation.cs

This interface will be implemented by each of the classes representing custom operations. The interface will contain a single method called “Execute” that will have a float array as its only input, and a float value as its only output.

In the final application, custom operations will be represented by the Unity-specific ScriptableObject class. It is anticipated that the class used for custom operations in the final application will implement the same ICustomOperation interface used in the prototype.

### InputValidator.cs

This static class is responsible for ensuring that incoming input meets the expectations of the operations that will use the input. It will contain several public methods that will test input against various criteria throughout the application. For example, it will contain a method that returns true if the input is greater than 0, which will aid in input error detection for the Pythagorean solve and inches-to-centimeters conversion custom operations.

In the final application, the Input Validator is expected to be a static class like the one created for the prototype.

### PythagoreanSolver.cs

This class is responsible for accepting a set of three inputs representing sides of a right triangle and calculating the length of the missing side (represented by a 0) of that triangle. The class will implement ICustomOperation. It will utilize the InputValidator.cs class to ensure its inputs are error-free. The class’s “Execute” method will read in a float array of length 3. It will interpret the float at index 0 as “a”, the float at index 1 as “b” and the float at index 2 as “c”. It will expect exactly one of these inputs to be 0, representing the missing side for which the solver must calculate a length. It will return a string representation of a float value representing the length of the missing side.

In the final application, custom operations will be represented by the Unity-specific ScriptableObject class. It is anticipated that the logic in this class remains the same between prototype and production, however in the production class there will likely need to be more data fields provided to be read by the frontend.

### FactorialOperation.cs

This class is responsible for accepting a single input (n) representing a number to apply the factorial operator (!) to and returning the factorial of the input (n!). The class will implement ICustomOperation. The class will implement ICustomOperation. It will utilize the InputValidator.cs class to ensure its inputs are error-free. The class’s “Execute” method will read in a float array of length 1. It will interpret the float at index 0 as the number to which to apply the factorial operator. It will convert the incoming float to an integer, as one can only apply the factorial operator to an integer. It will return a float representing n!.

In the final application, custom operations will be represented by the Unity-specific ScriptableObject class. It is anticipated that the logic in this class remains the same between prototype and production, however in the production class there will likely need to be more data fields provided to be read by the frontend.

### MetricConverter.cs

This class is responsible for accepting a single input representing a length in inches and returning the length converted to centimeters. The class will implement ICustomOperation. The class will implement ICustomOperation. It will utilize the InputValidator.cs class to ensure its inputs are error-free. The class’s “Execute” method will read in a float array of length 1. It will interpret the float at index 0 as the number to which to convert to centimeters. It will return a float that depicts the result of the conversion.

In the final application, custom operations will be represented by the Unity-specific ScriptableObject class. It is anticipated that the logic in this class remains the same between prototype and production, however in the production class there will likely need to be more data fields provided to be read by the frontend.

### Calculator.cs

This class is responsible for reading user input from the console, performing primitive operations or custom operations on received input, and printing the results of the calculations to the console. It will store the current value as an operand to be used in future calculations. It will have a single method called “Calculate” which will read in an array of strings from the console for processing. For the prototype, the Calculator will have a static list of tokens used to determine which kind of operation should be performed. This token must reside at index 0 in the string array. The tokens that correspond to each of the custom operations are as follows:

* “P” – Pythagorean Solve
* “F” – Factorial Operation
* “M” – Metric Converter

Based on which custom operation is selected, the calculator will be responsible for ensuring all inputs are appropriate for the ensuing calculation, making use of the InputValidator.cs class where necessary and implementing its own error detection logic where necessary.

Recall that error detection for operands with multiple decimal points, leading zeroes, and consecutive operators are to be handled in real-time in the final application (as users will be prevented from even entering invalid data). In the prototype, however, these errors will need to be detected after the user input is scanned in. Therefore, the calculator prototype will have error detection logic that will not be necessary in the final product. The error handling behavior in the prototype will be like the error detection described in the [Errors Resulting from Custom Operations](#_Errors_Resulting_from) section.

In the final application, there will be a frontend GUI through which the user will interact with the backend calculator. This will alleviate the need to read data from the console. Additionally, the ScriptableObject implementation of the custom operations will streamline the selection of the custom operation to execute. Additional logic will be necessary for the GUI to communicate with the backend (real-time input validation, for example).

## Error Handling in the Prototype

In lieu of communicating with a frontend GUI, the prototype will rely on the exception infrastructure in C#. Several custom exceptions will be created and stored in a single file, called CalculatorExceptions.cs. Each exception will derive from System.Exception and have four constructors as prescribed by the .NET C# documentation[[8]](#footnote-8). Those exceptions are as follows:

* Calculator Exception – A parent class for all custom exceptions for the calculator application. Instead of try-catching each individual exception type, the main program can try-catch calculator exceptions and handle them by displaying the syntax error and resetting the calculator input.
* Invalid Input Exception - An exception thrown when inputs passed to the calculator’s evaluation methods, or a custom operation are invalid. Examples include mismatched parenthesis or leading zeroes.
* Invalid Triangle Exception - An exception thrown when trying to perform Pythagorean operations on a right triangle that cannot exist.
* Invalid Expression Exception – An exception thrown when an infix expression is passed to the calculator that cannot be evaluated.
* Invalid Token Exception – An exception thrown when an expression passed to the calculator contains an invalid token (i.e., a symbol that is not an operator, or a letter.)

## Unit Testing

Unit tests will be designed for each class in the prototype using Visual Studio Code’s unit testing infrastructure. Unit tests will be also designed for each class in the final application that does not derive from the Monobehaviour class using Unity’s test runner package. If necessary, play tests will be designed for each class in the final application that derives from the Monobehaviour class using Unity’s test runner package.

# UI/UX Design

The final application will be created in Unity. All GUI elements will be created in this environment. The application will be visually comparable to other virtual calculators. These calculators are designed visually to simulate physical calculators.

Some prominent UI/UX features of the virtual calculators are large buttons, simple color schemes, and a display screen oriented at the top of the application. Buttons are large enough to be pressed with a finger in mobile versions of the application. As the user enters numerical values and primitive operators, they are shown in order of selection on the display screen. When the “equals” button is selected, the primitive operation is evaluated, and the resulting value replaces the operation string on the display screen. The user can then execute further primitive operations on the resulting value.

In addition to this basic setup, this application will feature a dropdown menu located between the display screen and the number pad. The dropdown menu will display all available custom operations. When a custom operation is selected, a dialog box will appear and prompt the user for necessary arguments. The user can then enter the arguments and execute the custom operation. If the operation is executed the current calculator input is cleared and the result of the custom operation will be displayed. Further primitive operations can be performed on this result. If the custom operation is cancelled, the current calculator input will remain.

This application’s calculator will also feature a console display located at the bottom of the virtual calculator that will be updated with error strings and other helpful information, to help the user better understand errors.

1. Source: <https://en.wikipedia.org/wiki/Order_of_operations> [↑](#footnote-ref-1)
2. Source: <https://runestone.academy/ns/books/published//pythonds/BasicDS/InfixPrefixandPostfixExpressions.html> [↑](#footnote-ref-2)
3. Source: <https://runestone.academy/ns/books/published//pythonds/BasicDS/InfixPrefixandPostfixExpressions.html> [↑](#footnote-ref-3)
4. Source: <https://en.wikipedia.org/wiki/Strategy_pattern> [↑](#footnote-ref-4)
5. Source: <https://en.wikipedia.org/wiki/Factorial> [↑](#footnote-ref-5)
6. Source: <https://en.wikipedia.org/wiki/Pythagorean_theorem> [↑](#footnote-ref-6)
7. Source: <https://en.wikipedia.org/wiki/Factorial> [↑](#footnote-ref-7)
8. Source: https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/exceptions/creating-and-throwing-exceptions [↑](#footnote-ref-8)