Calculator Technical Documentation

Programming III – Portfolio Activity 1.7

Alec Hislop

Contents

Data Structures

The calculator program has few runtime objects in memory and so has a simple data structure. There are only two double-type variables entered by the user from which all data processing and calculations are derived.

There are, however, many other variables used by the program to allow for full functionality and error trapping.

Below is a table of the data structure and the context for each variable:

Name	**Context**	**Type**	**Purpose**
btnPlusClicked	Calculator	bool	Boolean whether or not the Plus or Addition function button was the last button selected.
btnMinusClicked	Calculator	bool	Boolean whether or not the Minus or Subtraction function button was the last button selected.
btnDivideClicked	Calculator	bool	Boolean whether or not the Division function button was the last button selected.
btnMultiplyClicked	Calculator	bool	Boolean whether or not the Multiplication function button was the last button selected.
btnInverseClicked	Calculator	bool	Boolean whether or not the Reciprocal Inverse function button was the last button selected.
btnSquareRootClicked	Calculator	bool	Boolean whether or not the Square Root function button was the last button selected.
btnCubeRootClicked	Calculator	bool	Boolean whether or not the Cube Root function button was the last button selected.
btnTanClicked	Calculator	bool	Boolean whether or not the Tangent function button was the last button selected.
btnSinClicked	Calculator	bool	Boolean whether or not the Sine function button was the last button selected.
btnCosClicked	Calculator	bool	Boolean whether or not the Cosine function button was the last button selected.
temp	Calculator	double	Validated, a temporary user input used as the first number in arithmetic functions (e.g. *temp* ÷ other)
input	Calculator	double	User's input, validated, used as the input for trigonometric and algebraic functions, or the second number in arithmetic functions (e.g. other ÷ *input*)
output	Calculator	double	The calculated output of a function. Displays in the textbox.
outputDouble	Calculator.Parser()	double	Validated value from user's input, assigned to either *temp* or *input* above.
inputString	Calculator.Parser()	string	Non-validated user input as a string. The string is then parsed to ensure it is valid.
canParse	Calculator.Parser()	bool	Boolean value whether or not the inputString can be interpreted by the program. *True* if it can be parsed as a double. *False* if it cannot be parsed.

Algorithms

Error-handling techniques

There are a number of error-handling techniques used in the program to ensure that the user receives expected results.

The first error-handling technique is preventative. It stops the user from inputting invalid or malformed numbers. It does this by reading keystrokes that are entered into the program and rejects keys that are invalid. Invalid keys include most letters and symbols. The program accepts all base-10 digits, a maximum of 1 decimal point, and e-notation letters and symbols (e+ and e-).

The next error-handling technique will attempt to convert the user's input from string to double. If it cannot parse the input, the value is considered 0 (zero). This is done before any math library methods are called and ensures that only valid numbers are used with any function.

Finally, there are a few error traps implemented in the trigonometric functions. These are built into the system to ensure expected results are returned despite the limitations in floating-point mathematics when dealing with irrational numbers (Pi). For these trigonometric functions, input is received as degrees and converted to radians. The result is then calculated with this radian conversion. On certain inputs, the program will return *Undefined* or *Not a Number* (NaN). On other inputs, the program will return 0, 1 or -1. These are used when the expression is Tan(90°), Cos(180°) etc.

Pseudocode

Below is pseudocode for each method in both the Calculator class and each library.

Number buttons

```
btn0_click()
{
        Textbox.append("0")
}
btn1_click()
{
        Textbox.append("1")
}
btn2_click()
{
        Textbox.append("2")
}

[etc.]
```

Basic calculator methods

```
// Checks the input if it's a valid number
Parser(String input)
{
	Bool canParse = TryParse(input, out output)

	If canParse == true
		Return output
	else
		Return 0
}

// Resets all button presses and numbers, clears textbox
btnCancel_click()
{
	allButtonsClicked = false
	temp = 0
	input = 0
	textbox.text = ""
}

// When the equals button is pressed, the program checks which operation is requested and uses the
appropriate library.
btnEquals_click()
{
	Input = Parser(textbox.string)
	If btnPlusClicked == true
		Output = Arithmetic.Add(temp, input)

	Else if btnMinusClicked == true
		Output = Arithmetic.Subtract(temp, input)

	[...]

	Else if btnSquareRootClicked == true
		Output = Algebraic.SquareRoot(input)

	Else if btnCubeRootClicked
		Output = Algebraic.CubeRoot(input)

	[...]

	Else if btnTanClicked == true
		Output = Trigonometric.Tan(input)

	Else if btnSinClicked == true
		Output = Trigonometric.Sine(input)

	[...]

	Textbox.text = output
}

// Validating keystrokes in textbox
Textbox_KeyPress(KeyPress e)
{
	// Checks that the character is a digit, decimal, or e-notation
	If e = isDigit or
	e = '.' or
	e = 'e' or
	e = '+' or
```

```
        e = '-'
        {
                [Accept KeyPress]
        }
        Else
        {
                [Reject KeyPress]
        }

        // Checks that there is a maximum of one decimal and e-notation symbol
        If e = '.' && '.' count > 0
                [Reject KeyPress]

        If e = 'e' && 'e' count > 0
                [Reject KeyPress]

        If e = '+' && '+' count > 0
                [Reject KeyPress]

        If e = '-' && '-' count > 0
                [Reject KeyPress]
}
```

Arithmetic buttons

```
btnMinus_Click()
{
        btnMinusClicked = true
        [all other buttons] = false

        temp = Parser(textbox.Text)
        textbox.text = ""
}
btnMultiply_Click()
{
        btnMultiplyClicked = true
        [all other buttons] = false

        temp = Parser(textbox.Text)
        textbox.text = ""
}
btnDivide_Click()
{
        btnDivideClicked = true
        [all other buttons] = false

        temp = Parser(textbox.Text)
        textbox.text = ""
}

[etc.]
```

Trigonometric and algebraic buttons

```
btnSquareRoot_Click()
{
	btnSquareRootClicked = true
	[all other buttons] = false
}

btnCubeRoot_Click()
{
	btnCubeRootClicked = true
	[all other buttons] = false
}

btnSin_Click()
{
	btnSinClicked = true
	[all other buttons] = false
}

[etc.]
```

Arithmetic functions

```
Add(double a, double b)
{
	return (a + b)
}

Subtract(double a, double b)
{
	return (a - b)
}

Divide(double a, double b)
{
	return (a / b)
}

Multiply(double a, double b)
{
	return (a * b)
}
```

Trigonometric functions

```
Tan(double a)
{
	double radians = a * (PI / 180)
	double result = Tan(radians)

	if (radians = PI / 2)
		return NaN
	if (radians = -PI / 2)
		return NaN
	return result
}

Sine(double a)
{
	double radians = a * (PI / 180)
	double result = Sin(radians)
```

```
        if (radians = PI / 2)
                return 1
        if (radians = PI)
                return 0
        return result
}

Cosine(double a)
{
        double radians = a * (PI / 180)
        double result = Cos(radians)

        if (radians = PI / 2)
                return 0
        if (radians = PI)
                return -1
        return result
}
```

Algebraic functions

```
SquareRoot(double a)
{
        Return Sqrt(a);
}

CubeRoot(double a)
{
        Return Pow(a, (1.0 / 3.0));
}

Inverse(double a)
{
        return (1.0 / a);
}
```

Testing procedure

The program should be thoroughly tested before commercial release. It is recommended that the program undergo unit testing, followed by integration testing, and finally system testing before being released.

Unit testing

The program will be unit tested within each class. The program has 4 principal classes that need to be tested in isolation:

- Calculator
- Algebraic
- Trigonometric
- Arithmetic

The `Algebraic`, `Trigonometric` and `Arithmetic` classes each have 3 or 4 methods. Each of these methods needs to be set up to be tested without any integration with other components. To do so, the methods could be tested in a separate environment and with hard-coded input to ensure that the output is appropriate.

For example, the `Arithmetic` class contains methods for performing basic arithmetic operations: addition, subtraction, multiplication and division. To test the `Arithmetic.Add(double a, double b)` method, the two parameters are hardcoded and the output is checked and logged. A similar process is used in unit testing the other classes and methods.

This testing could be done manually, or the test could be set up to be automated, testing many inputs at once. Logging could be done with a simple test table in a spreadsheet.

Integration testing

Once unit testing is complete, integration testing commences. This examines how the classes behave together in a broader context than unit testing. For this program, the `Calculator` class is tied closely with the `Algebraic`, `Trigonometric` and `Arithmetic` classes. Integration testing ensures that that these classes (and their respective methods) correctly work together.

The `Calculator` class handles user input and, based on what the user does, this input is used as a parameter for the various methods in other classes. To test this integration, an environment is set up where the input could be automated and many class relationships could be tested at once. For example, the `Calculator` class generates numbers for the `Trigonometric.Sin(double a)` method and the output is checked and logged in a spreadsheet. Should any unexpected behaviour occur, it would be known that the issue is in integration – that is, the methods have already been unit tested in isolation, so the fault is in the integration process.

System testing

The entire program can then be tested from end-to-end. This testing examines the whole system, involving both functional and non-functional aspects. This is typically done to evaluate how well the system fulfils the design requirements and uses black-box testing techniques to approximate how the software would be used in a live environment.

To set up a system test, the environment needs to be as close to a live environment as possible. This includes things like using the same operating system as the client and using comparable hardware and I/O devices.

Some of the functional aspects that would be tested include:

- All mathematical processes work as the user expects.
- All UI buttons are legible, unambiguous and work as expected.
- A broader regression test, to ensure any changes made previously haven't altered the behaviour of other components unexpectedly.

Some of the non-functional aspects that would be tested include:

- The performance of the software, how fast and responsive the application is, especially under stress or load.
- The usability of the software, how easy it is to operate and interact with it and any UI components that may need to be redesigned.
- The compatibility of the software with hardware and operating systems of the target environments.

Upgrades and Future Enhancements

The program is currently feature-complete and functional (v1.00) according to the scope of this original development lifecycle. However, that does mean that the scope for the program could not be revised and expanded in later iterations.

There are a number of limitations in the current version, such as a lack of user feedback for expressions, lack of certain constants (e.g. a Pi button) and extended mathematical functionality (e.g. brackets, exponents, log etc.).

Moreover, the program may not be sufficiently robust enough for smooth user operation. Despite keystrokes being parsed almost instantaneously as they are entered, invalid input can still be entered through the mouse or other system input methods – e.g. the user can right-click the textbox and paste invalid or malformed numbers. Although these invalid numbers are caught and safely handled, it still appears rough and unpolished in regards to the user's experience.

Some potential features for later include:

- Handling of different angle units (radians, grads) and different number bases (hexadecimal, binary)
- Percentage button (e.g. input 10 + 5%, output 10.5)
- Additive inverse (e.g. input (+7) ±, output (-7))
- Factorial (e.g. input 4!, output 24)
- Exponents (e.g. input 2^4, output 16)
- Common irrational constants buttons (e.g. Pi, Euler's number, golden ratio etc.)

Some potential quality-of-life, user interface and user experience improvements include:

- Parsing and validating pasted clipboard text in real-time
- Multiple functions in a single expression (e.g. input 1 + 4 ^ 2, output 17)
- Function and output history
- Digit grouping (e.g. display 1 million as 1,000,000 or 1 000 000)
- Displayed expression before hitting equals (display the full expression, e.g. 2 + 3 in the textbox)
- Unit conversions between common metric, imperial, US, and SI units.