# 3. Development

## 3.1 Success Criteria and Testing

### 3.1.0 Version 0.1.0

#### V0.1.0 – Success Criteria

In Version 0.1, I attempted to complete the success criteria:

* “Functionality for the four basic operators, +, -, \*, ÷.”
* “Follow the rules of BODMASS.”
* “Have functionality for using square roots and powers.”

This version covers the success criteria that make up the most basic functionality that a calculator.

For Version 0.1.0 to be successful, the calculator should be able to calculate any basic equation that includes the four basic operators, “+, -, \*, ÷” and powers and roots.

Version 0.1.0 was started by creating an XML layout file which contained all the required parts of the calculator, such as the number input buttons, the calculator display, the operator buttons, a delete button, etc.

The layout was created using a *Constraint Layout*, instead of a *Relative* or *Linear layout*. A *Constraint Layout* is where each view has constraints that act as connections to other views, the parent layout or an invisible guideline.

A *Relative Layout* is where the views are positioned relative to each other, while a *Linear Layout* is where the views are organised into a single horizontal or vertical row.

I have chosen to use a *Constraint Layout* because although it is initially more complicated to understand and implement, you can build much more flexible layouts with it. This will be especially helpful in this project as the *Constraint Layout* will mean the layout will scale and change appropriately depending on different size devices.

In the *MainActvitity* class, variables were linked to their corresponding XML element. For example, the buttons were linked to Button variables and the calculator display was linked to a *TextView* variable.

The buttons variables were set an *OnClickListener* so they called the *onClick* method when tapped.

A switch case statement was used where cases are dependent on the ID of the clicked button. Each button has a unique ID that’s defined in the layout’s XML file.

Each button case adds the input of the button into the expression so it can later be used in the *ShuntingYard* class.

There are also special cases, such as the “AC” button which wipes all data, and the “DEL” button which removes the last character of the expression.

When the equals button is pressed (“=”), a validation is first run to ensure that the expression string is not empty.

This is done to prevent an error occurring when the string is passed to the *ShuntingYard* *infixToPostfix* method.

If the validation check is failed, the stack trace is logged and an error message is shown to the user.

If the validation check is successful, the answer is passed to the *ShuntingYard* method called *infixToPostfix*. This method converts the user’s expression into Reverse Polish Notation.

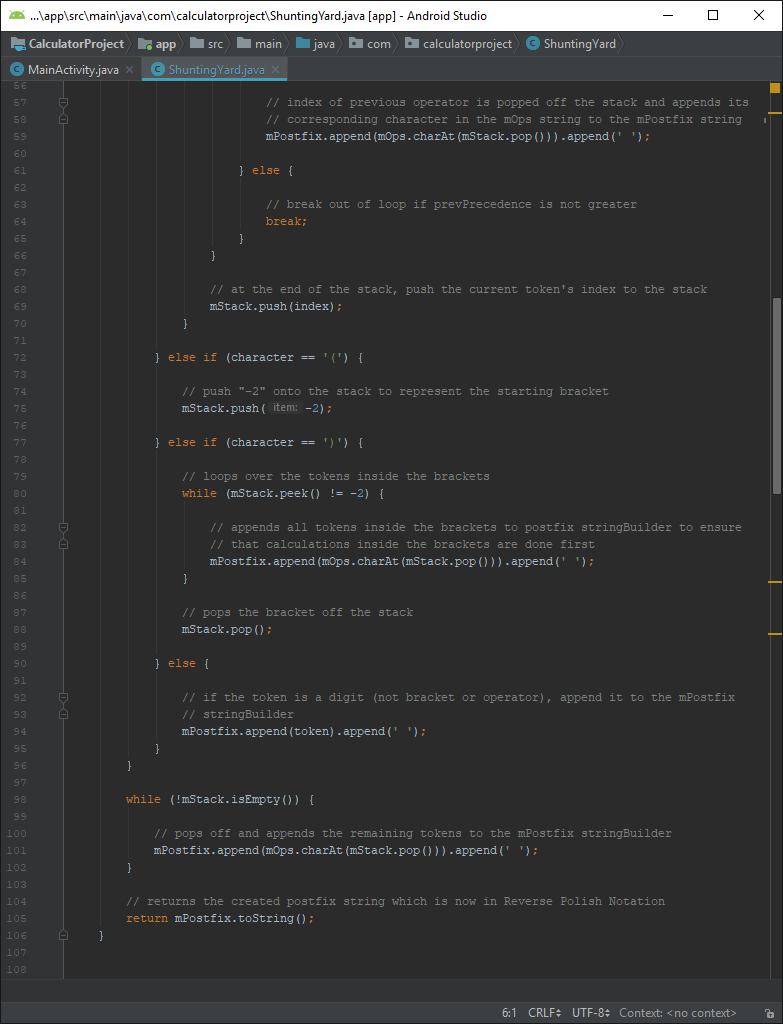
In this method, a string called *ops* is used to define each operator’s precedence. The precedence is found by the index of the operator divided by 2.

A for-each loop is used to iterate through the user’s expression by each character. If the current character is empty, a *continue* keyword is used to terminate the processing of the current iteration and continue with the for-each loop.

Variables are made to denote the current character and the index of the character in the operator string. If the character is not an operator, it will not exist in the string and therefore getting its index will return “-1”.

If the index is returned as “-1”, and the stack is currently empty, the value of the index is pushed onto the stack.

If the stack is not empty, a while loop is used so that while the stack is not empty, the value of the precedence before the current character is retrieved by using “*stack.peek() / 2”*. The *peek* method is used to get the value of the top of the stack. This will be the index of the previous character in the *ops* string and dividing it by two returns the precedence value. This is stored in a variable called *prec2*.

The precedence of the current character is calculated by dividing the index of the character in the *ops* string by 2. This is stored in a variable called *prec1**.*

If the previous operator’s precedence is greater than the second one or they are equal (if the current operator is not a power), the index of the previous operator is popped off the stack and its corresponding operator character is appended to the postfix string.

When the while loop ends (i.e. the expression is empty), the current index is pushed onto the stack.

If the index does not equal “-1”, and the character equals an opening bracket “(“, the value of “-2” is pushed onto the stack to represent the start of the bracket.

If the current character is a closing bracket instead, a while loop is used to add characters to the postfix string. The condition for the while loop is while the previous element in the stack does not equal “-2” (the integer representing an opening bracket).

This means that the part of the expression inside the bracket will take precedence over everything else and therefore obey the rules of BODMAS properly.

If none of these conditions (index != -1, character = “(“ or “)”) are met i.e. the current character is just a digit, the digit is appended to the postfix string.

Finally, the *infixToPostfix* method returns the string it has made.

The source code that handles this is shown below:



In the *MainActivity* class, the postfix can be found by calling the *infixToPostfix* method on the user’s input expression string.

The *evaluateRPN* method can then be used on the postfix string to calculate the answer to the user’s expression.

The *evaluateRPN* method works by first creating a stack with a data type of double called “tokens” and then running a for-each loop where the loop iterates over an array created from the postfix that has been split up on the whitespace.

Each iteration, the method uses a HashMap made using a for-each loop which goes over the *enum* S*ign* which contains all the operators and puts each operator into the HashMap.

The *evaluateRPN* can then use the *find* method on the *HashMap* with the parameter of the current token to see if it is an operator. If the token is not an operator, it will return “null”.

If the token is an operator, the operator is applied to the last two elements of the stack using the *calcSign* method and the *enum* data type (which contains all the operators and returns the result for each one).

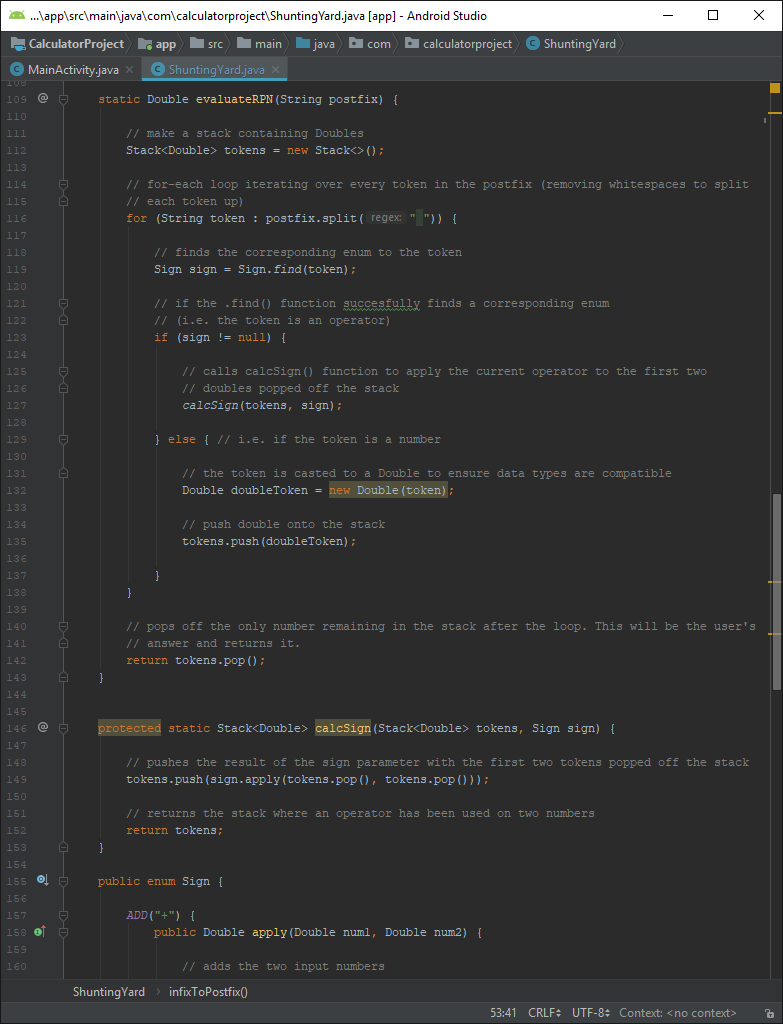
The *calcSign* method returns the stack after the tokens have been popped off, the result of the two with the operator has been calculated and then the result has been pushed back on.

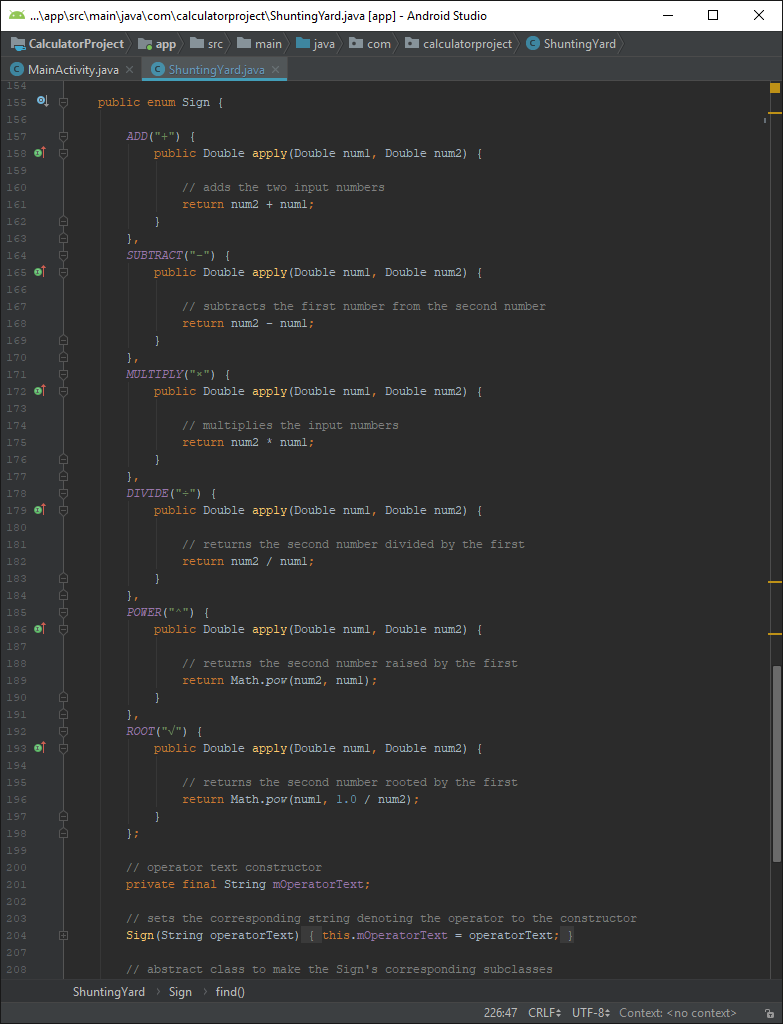
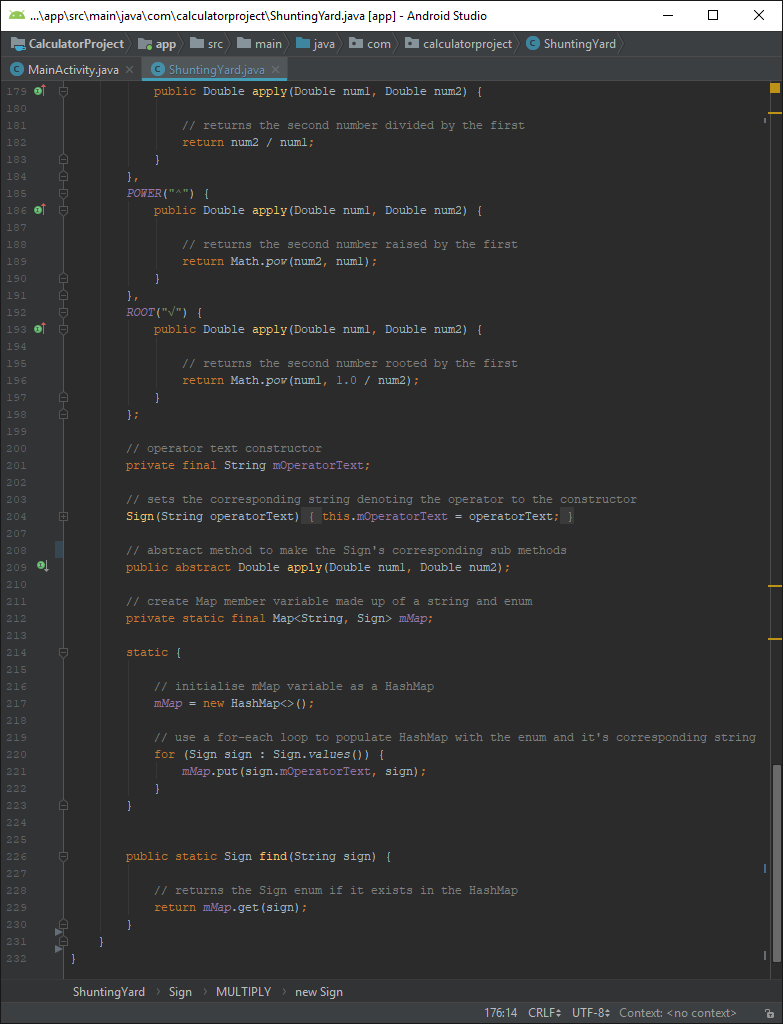
If the current token is not an operator, it must be a number and therefore the token is converted from a string to a double and then pushed onto the stack.

At the end of the iteration, the only thing left in the stack will be a single double data type which is the result of the user’s input expression.

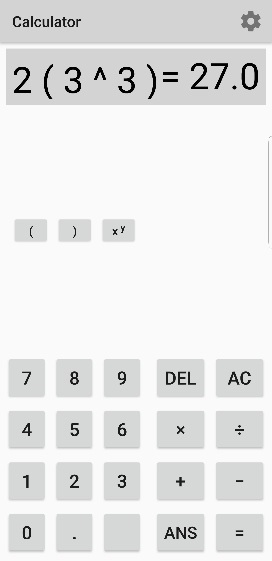
This value is returned where it can be displayed to the user using a *TextView*.

The source code that handles this is shown below:



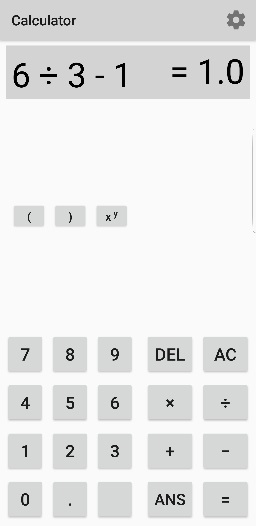
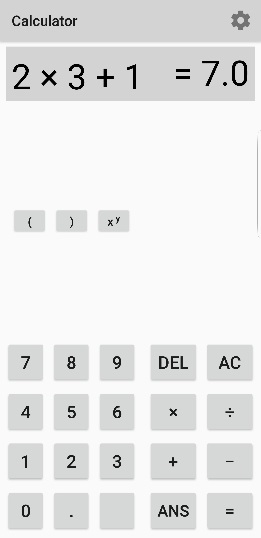


Version 0.1.0:



#### V0.1.0 - Testing

The first of these success criteria is the four basic operators:

These screenshots show how the calculator properly handles the operators “+, -, x, ÷”. The input takes “2\*3+1” and outputs “7”. This is correct and therefore shows that the calculator is handling multiplication and addition correctly.

The calculator also takes “6÷3-1” and outputs “1”. This is the correct result and there for shows that the calculator is performing division and subtraction correctly.

These screenshots also show that the calculator is properly following BODMAS when completing the calculations. This is because when inputted with “2\*3+1”, the calculator first does “2\*3” and then “+1”.

This is correct because multiplication takes precedence over addition in BODMAS. Therefore, when using the basic operators BODMAS is correctly used. This means that the first and second success criteria aimed to be completed in Version 0.1 were successful.

The last success criteria in Version 0.1 is for the calculator to “have functionality for using square roots and powers”.

These screenshots show how Version 0.1 of the calculator handles powers and how they interact with BODMAS.

In the first screenshot, the calculator interprets the equation as “2x33”. Therefore it follows BODMAS and calculates “33” to give “27” because powers have precedence over multiplication. It then multiples “27” by 2 to result in “54”. This is an example of the calculator working correctly.

However, in the second screenshot the equation “2(33)” is inputted. This is using a different way of expressing the exact same equation as the first screenshot.

The calculator should be interpreting this expression as “2\*(33)”. However, the calculator does not recognise a number in front of brackets as shorthand for “2\*(33). Therefore, the base and power does not get multiplied by two and the output is incorrect.

Another problem with version 0.1 is that the functionality for roots was not completed in the first iteration of the iterative development process. However, this can be resolved in the next iteration.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Reason | Input | Expected | Actual | Required Actions |
| Test basic operators and how they work with BODMAS. | 2\*3+1 | 7  =(2\*3)+1 | 7 | N/A |
| Test basic operators and how they work with BODMAS. | 6/3-1 | 1  =(6/3)-1 | 1 | N/A |
| Test that powers are calculated correctly. | 33 | 27 | 27 | N/A |
| Test BODMAS precedence in an expression with different operators but no brackets to simplify it. | 2\*4 -22 | 4  =(2\*4)-(22) | 4 | N/A |
| Test how brackets are handled when used as shorthand for multiplication. | 2(33) | 54  =2\*(33) | 27  =(33) | Calculator does not recognise a number next to a bracket as shorthand for multiplication and instead ignores the number.  Actions:  When interpreting the user’s input expressions, insert multiplication operators where needed before evaluating the expression. |
| Test that square root function works correctly. | sqrt(4) | 2 | - | Root functions have not yet been created.  Actions:  Create a function that takes two inputs, the base and the root. It will root the base by the root variable and output the result to the user. |
| Test “DEL” button to make sure it deletes one operator/number at a time | Tap “DEL” on any expression.  E.g.:  2-2+ | 2-2 | 2-2+ | When the user inputs operator characters, the application automatically inserts “ “ characters either side of the user’s input. This is done so the Reverse Polish Notation is created correctly.  The “DEL” input does not include the whitespace that was inserted by the application. To delete this as well, the user needs to input “DEL” multiple times.  Actions:  Currently, the “DEL” button works by removing the last character of the expression string.  To fix this, first remove all whitespace in the string.  Then remove the last character.  Finally, the whitespace can be inserted back into the string. |
| Test “AC” button to ensure that it deletes the entire expression and wipes anything in the calculator’s memory that should be non-volatile. | Click “AC” on expression:  2+3 | Wipes current display and everything stored in memory. | Wipes current display and everything stored in memory. | N/A |

### 3.1.1 Version 0.1.1

#### V0.1.1– Success Criteria

V0.1.1 involves completing the success criteria for:

* “Have functionality for using … roots”
* Fix various bugs introduced in version 0.1.0 such as the *DEL* function behaving improperly and correcting how using brackets as shorthand for multiplication is handled.

This version covers the second iteration of the implementation of the first 3 basic success criteria.

This iteration is made up of fixing bugs that were introduced in the first iteration. This includes the errors where the “DEL” button was not working correctly because it only deleted the last character of the expression string variable.

This bug was fixed by removing all whitespace from the expression, then deleting the last character from the string. Then whitespace was inserted back into the string. This was done using the string function *.replaceAll* and the regex expression “*(‘.(?=.)’, ‘$0 ’)”.*

This regex expression replaces all characters in the string with itself and space except the last character.

In this iteration, the bug where using brackets as shorthand for multiplication was not interpreted correctly was also fixed.

This was fixed by creating a function that was called when an opening bracket was entered by the user.

This function checks if the character before it is a digit, e.g. “4(2)” or a closing bracket, e.g. “(2)(2)”. If the check is positive, a *StringBuilder* is made and a multiplication sign and the appropriate whitespace is inserted into the string. This is done using the *StringBuilder* method *replace*.

This iteration also included the implementation of a root function. This was created by adding a button to the main XML file and linking it to the *MainActivity* class so it would call *onClick* when the button is clicked. When the root button is clicked, it inserts the radical sign (√) into the expression string and updates the calculator display to show this.

The radical sign was then included in the *ShuntingYard* class to assign its precedence in BODMAS when creating the Reverse Polish Notation. This ensures that the RPN is formed correctly so BODMAS is followed.

Extra parameters were also created to calculate the user’s inputted roots. The roots are calculated by using Java’s Math library: “*Math.pow(base, 1.0 / root)”*. This is because mathematically √4 is equal to 41/2 through indices laws.

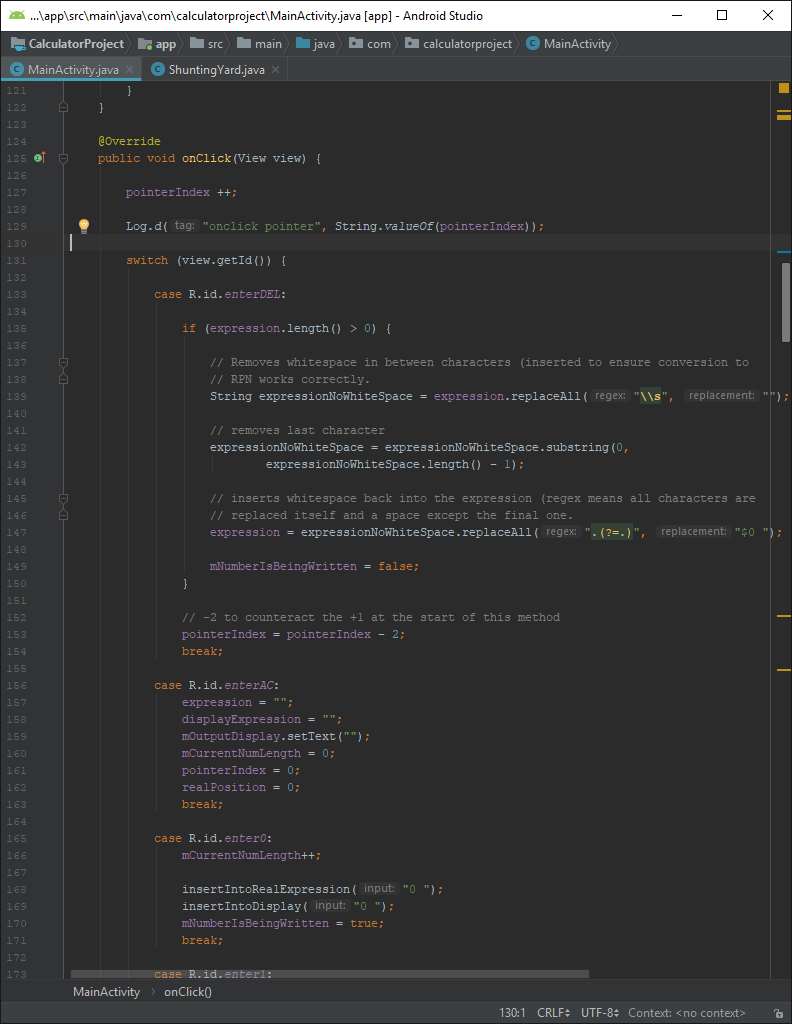
Version 0.1.1 also included the addition of shortcut root and power buttons.

The shortcut power button automatically adds 2 to the user’s base so the user does not need to enter their power as well.

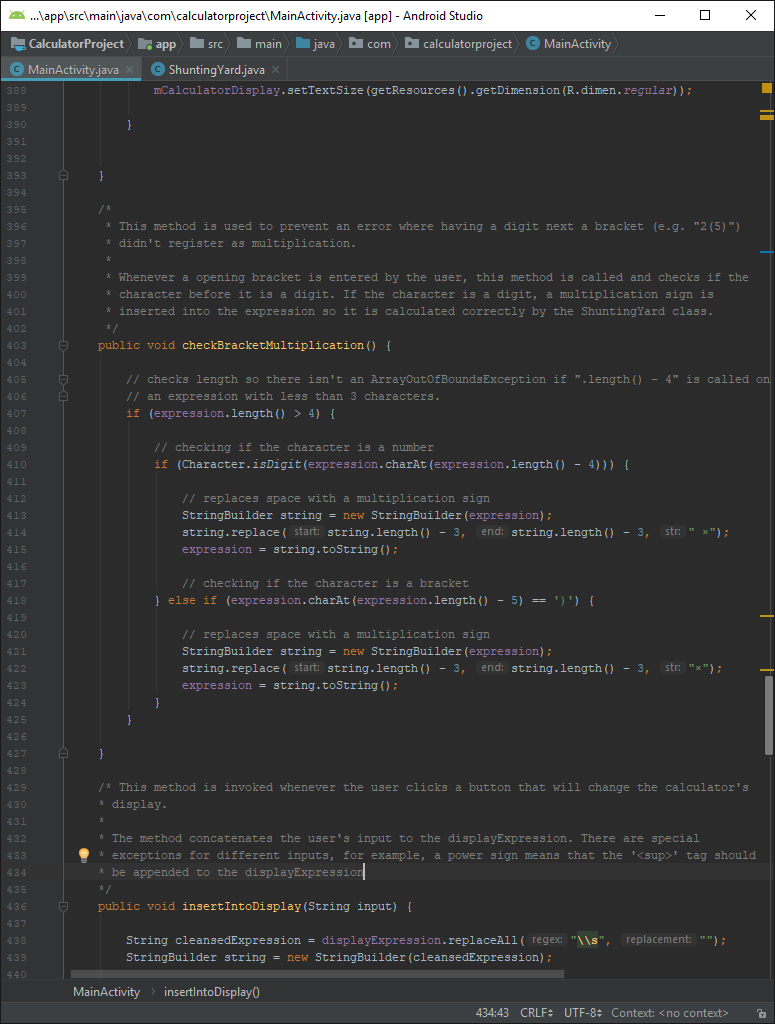
Squaring numbers is the one of most common power used by students and therefore having a shortcut button makes the process of using the calculator much more efficient. This is important as it leaves the user to focus and spend their time and focus on other things.

The square root shortcut button acts similarly to this, it automatically adds a square root because square roots are the most common roots used.

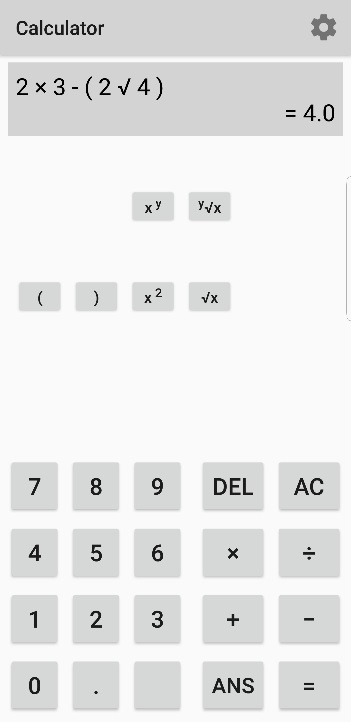
The source code showing the improvements to the *DEL* button:



The source code correcting bracket shorthand:



Version 0.1.1:



Another notable change in this version is that the size of the text in the calculator display was decreased to ensure that the user’s entire expression and answer will be displayed clearly to the user.

#### V0.1.1 – Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Reason | Input | Expected | Actual | Required Actions |
| Test “DEL” button to make sure it deletes one operator/number at a time | Tap “DEL” on any expression.  E.g.:  2-2+ | 2-2 | 2-2 | N/A |
| Test how brackets are handled when used as shorthand for multiplication. | Input an expression where there’s a number adjacent to an opening bracket, e.g.:  3(33) | 3(33)  = 54 | 3\*(33)  = 54 | N/A |
| Test that x/y root function works correctly. | Input any y√x expression to test that the root and base are working correctly. | 3√8  = 2 | 3√8  = 2 | N/A |
| Test square root shortcut button works correctly. | Use the shortcut button to input any 2√x expression. | 2√4  = 4 | 2√4  = 4 | N/A |
| Test square power shortcut button works correctly. | Use the shortcut button to input any x2 expression. | 23  = 8 | 23  = 8 | N/A |

### 3.1.2 Version 0.1.2

#### V0.1.2 – Success Criteria

Version 0.1.2 did not tackle any specific success criteria, however, this version focused on improving the appearance and ease of use with the app.

The appearance was improved in this version by creating a new string to display the user’s expression.

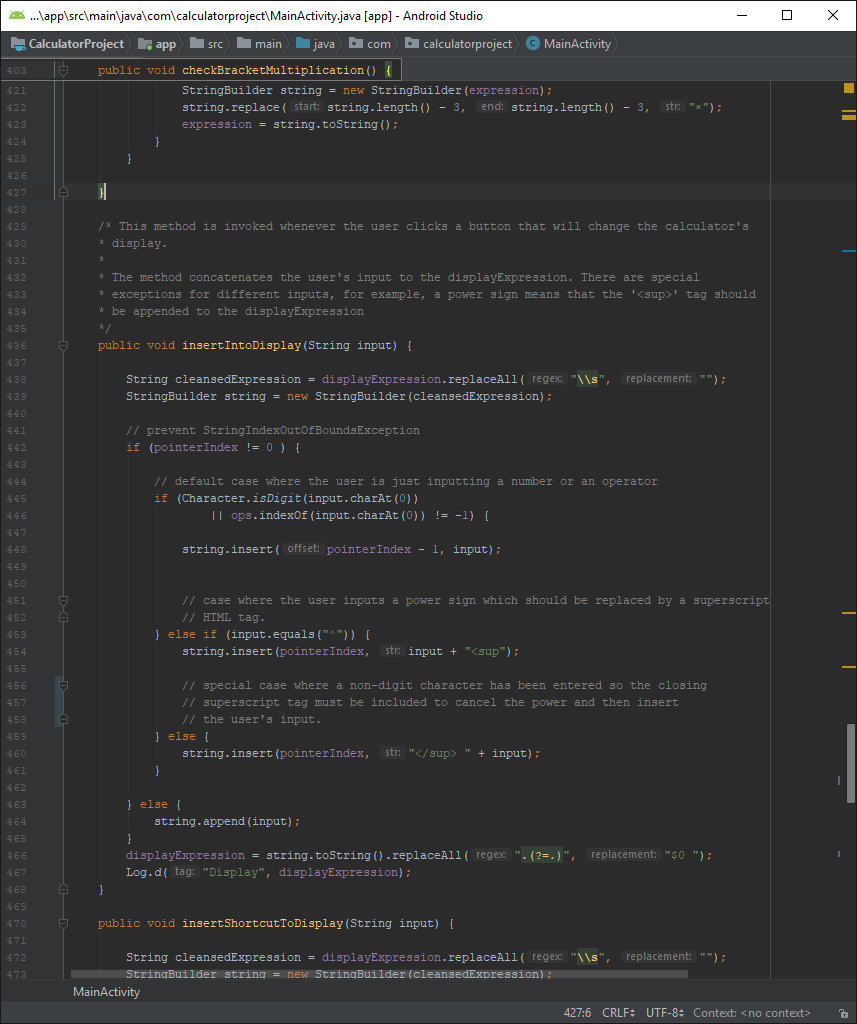
The original string acted as the input to the Reverse Polish Notation conversion method as well as being displayed to the user. This means the string uses operators such as “^” to denote powers. However, in mathematics powers are usually shown using superscript, e.g. “xy”.

For example, the user’s expression were displayed as “2 + 4 ^ 2” whereas they would normally be displayed as “2 + 42”.

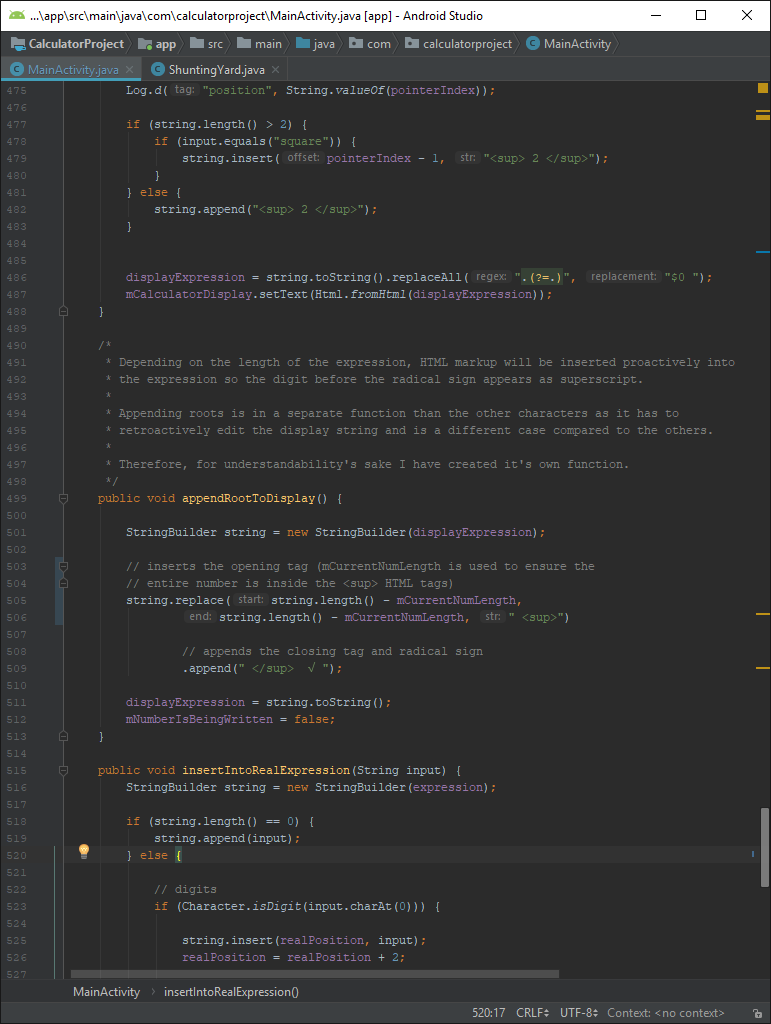
To improve the appearance of the display I created a different string that included HTML tags to apply mark up to the string. For example, when the user clicks the power button, the next number entered is surrounded by *<sup></sup>* tags. When a string with *<sup>* tags is applied such as:

“ *.setText(Html.fromHtml(‘2 + 4 <sup>2</sup>’))”*

The HTML is applied to the display string so it appears as “2 + 42”.



This change was also implemented for roots:



Implementing mark up for roots required a different function as the HTML tags had to be inserted retroactively into the string.

Version 0.1.2:



#### V0.1.2 – Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Reason | Input | Expected | Actual | Required Actions |
| Ensure powers are displayed correctly. | 3^2 | 32 | 32 | N/A |
| Ensure roots are displayed correctly. | 3√2 | 3√2 | 3√2 | N/A |
| Ensure multi-digit numbers as a power are displayed correctly. | 3^100 | 3100 | 3100 | N/A |
| Ensure multi-digit numbers as roots are displayed correctly. | 100√2 | 100√2 | 100√2 | N/A |

### 3.1.3 Version 0.2.0

#### V0.2.0 – Success Criteria

Version 0.2 is intended to complete the success criteria:

* “Cursor controls that will move the cursor around the equation.”

This means that for Version 0.2 to be considered successful, the app must have functionality that allows the user to traverse through the equation using cursor buttons. This will enable the user to edit their expression without either clearing it using *AC* or deleting part of it.

This was completed by introducing a variable that kept track of the user position in the expression’s string.

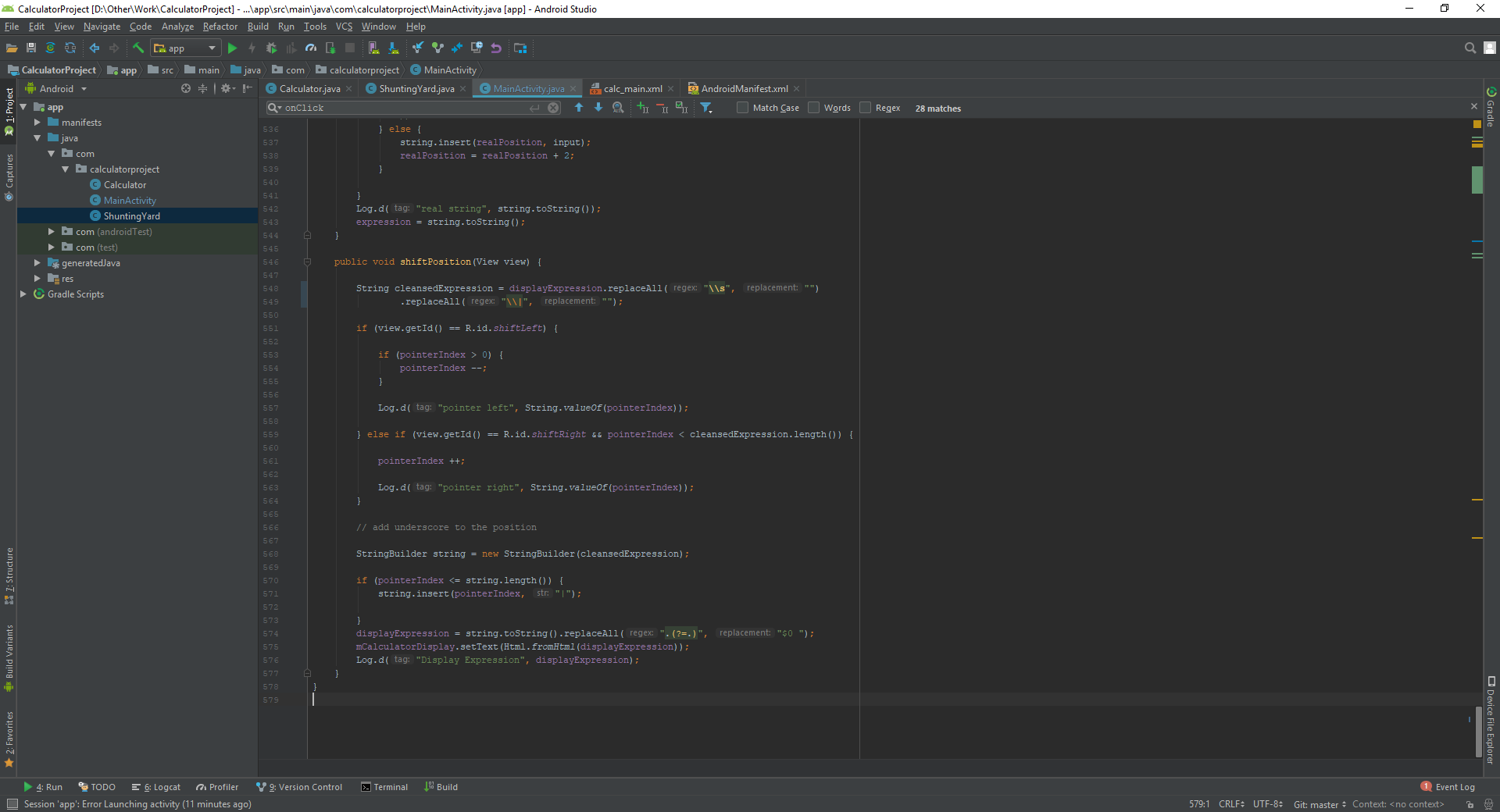
This variable was incremated every time the user added to the expression and was decreased whenever the user deleted something from the expression.

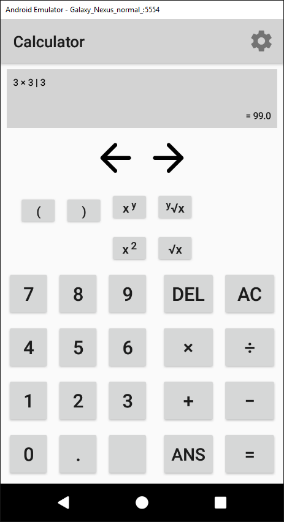
Two buttons were also added which were given the IDs of *shiftLeft* and *shiftRight*. They were both linked to a function that would either decrease or increase the position variable by one depending on which button was clicked.

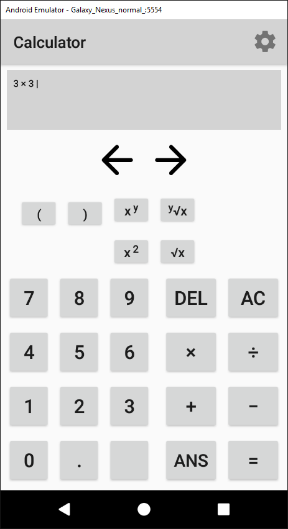
In this function, a vertical slash character (“|”) is also added to the string in the current position index so the user has a graphical representation of where the position index is.

To finish the implementation of this success criteria, the method that handles the insertion of the user’s input to the expression was changed from adding the input to the end of the string. Instead, it would use a StringBuilder and insert the user’s input at the position index.

The code that handles the user moving through their expression:



Version 0.2.0:



These screenshots show how the cursor can used to edit the expression by inserting a number and it will give the correct result.

#### V0.2.0 – Testing

Whilst testing Version 0.2.0, I noticed that the foundation of the project was not built as efficiently as it could have been, and this will affect the process of adding further features to the project.

The source of this inefficiency was located in the main calculator Java class which handled the user’s interactions and the GUI. The *ShuntingYard* class (which handles the process of converting the user’s input into an answer) did not experience these same problems.

The logic and structure of the code in the main activity was constructed improperly which created source code that was not easily understandable and therefore was difficult to maintain. This was especially clear as creating new features and adding them to the project became more complex as the project grew larger.

For example, each button the user could tap to enter input into the calculator was linked to an *OnClickListener* in the main activity. This overrode the android superclass *onClick* and handled each button separately using a switch-case statement. This was a very inefficient method to implement functionality for the buttons because it meant each button had to be programmed separately even when similar buttons did very similar things. E.g. tapping the “8” or “9” button does the same thing except enters a different digit.

Another effect of this illogical implementation is that the code being very long and complicated. This made maintaining the code almost impossible and making small changes to fix bugs became needlessly difficult.

Another example of how the codebase was built up incorrectly is a logical error that exists in the app. This logical error is that the variable that tracks the user’s position in the expression was being incremented at the wrong time in the program’s flow. This meant that when the position variable was used to insert input into the expression which often caused an off-by-one error.

This logical error was caused due to the lack of structure in the codebase which reduced the source codes readability and so lead to the code being difficult to understand and maintain.

In order to amend this, the next version will consist largely of refactoring the parts of the source code that handles the graphical user interface and the user’s input.

For example, I will streamline the process of the user entering input. I will remove the many separate buttons and instead of using *OnClickListeners* I will use the *onClick* XML characteristic which will call a specific method.

With this, I can group the buttons into similar groups and each set of groups will call their own method. For example, all the digit buttons (0-9) will call the *inputDigit* method and all operator buttons (+, -, ÷, ×, etc.) will call the *inputOperator* method.

I can use the *tag* XML characteristic to define which input (i.e. which digit or operator) to insert into the string depending on which button was clicked.

### 3.1.4 Version 0.3.0

#### V0.3.0 – Success Criteria