Design

Variables

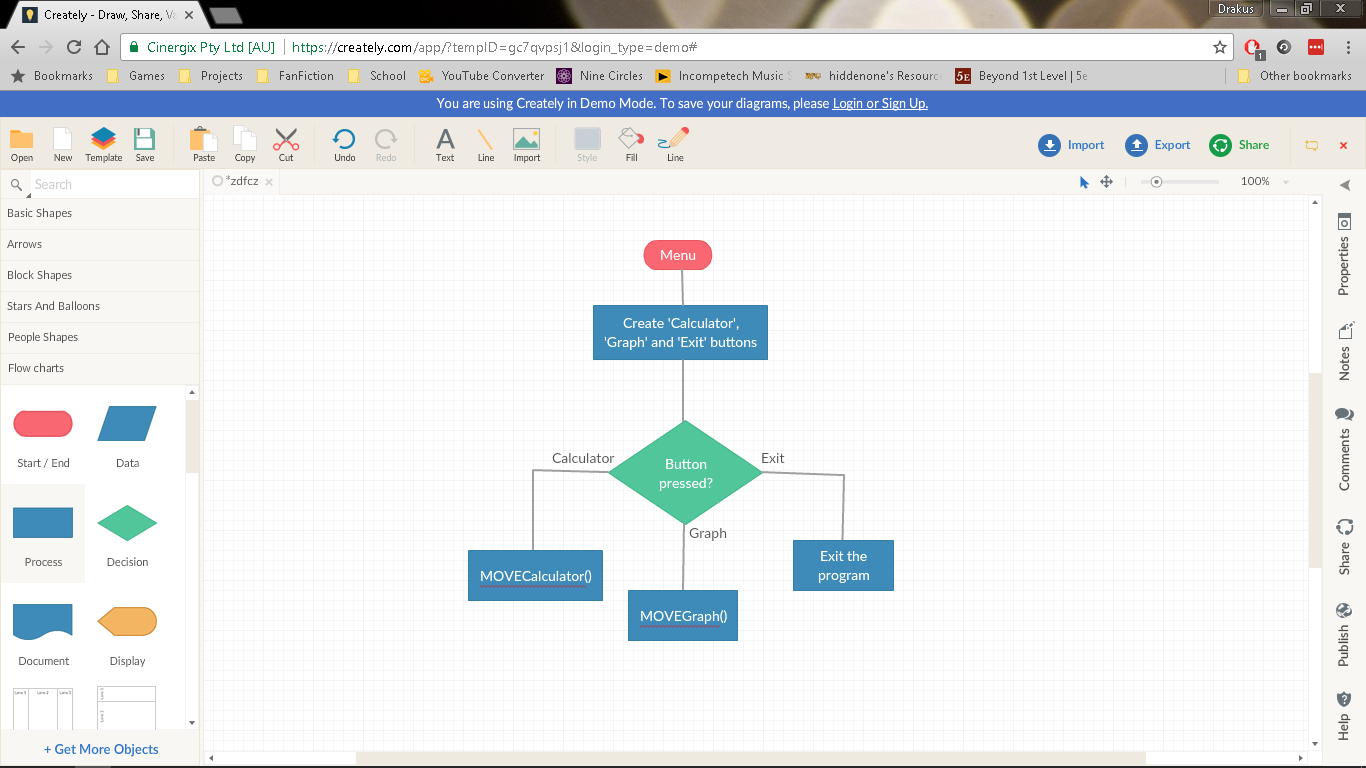
I have decided to name the main variables before I start the creation of the program.

|  |  |
| --- | --- |
| Name | Function |
| Term | Stores the current term that is to be added to the calculator’s expression. |
| Equation | Stores the equation to be answered and calculated. |
| CalculatorExpression | A formatted version of the Equation that will be displayed to the user on the calculator screen. |
| ExpList | The Equation variable, split into a list of characters that can be searched and altered by the Calculate function |
| e | Shorthand for the ExpList variable that will be used in the Calculate function. |
| x | Iterative Variable. Used in the calculator grid search and the graph drawing. |
| y | Iterative Variable. Used in the calculator grid search, and the graph drawing loop as a shorthand for the calculated value for each point. |
| i | Iterative Variable. Used in non-Cartesian lists. |

Dataflow and Algorithms

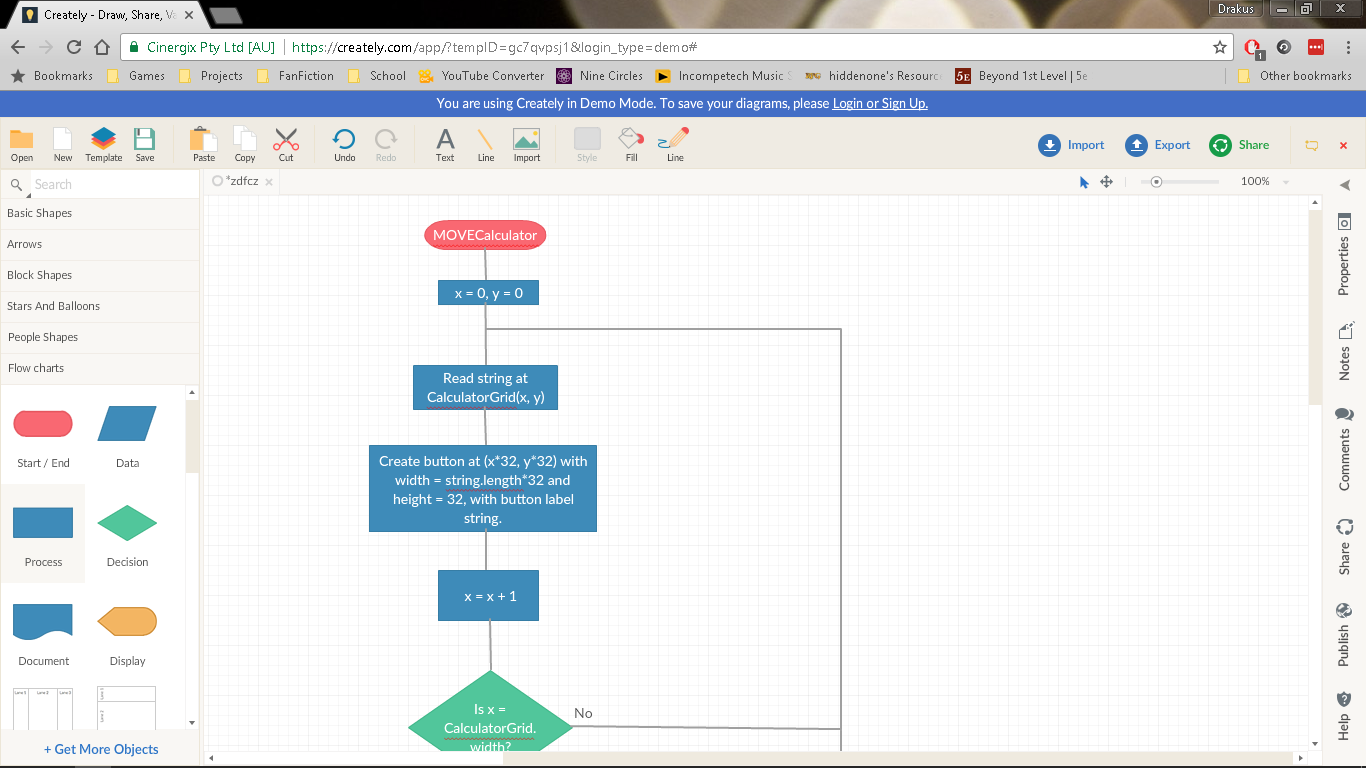
I have decided to name the main functions before I start the creation of the program, as it would make it easier to refer back to them when needed.

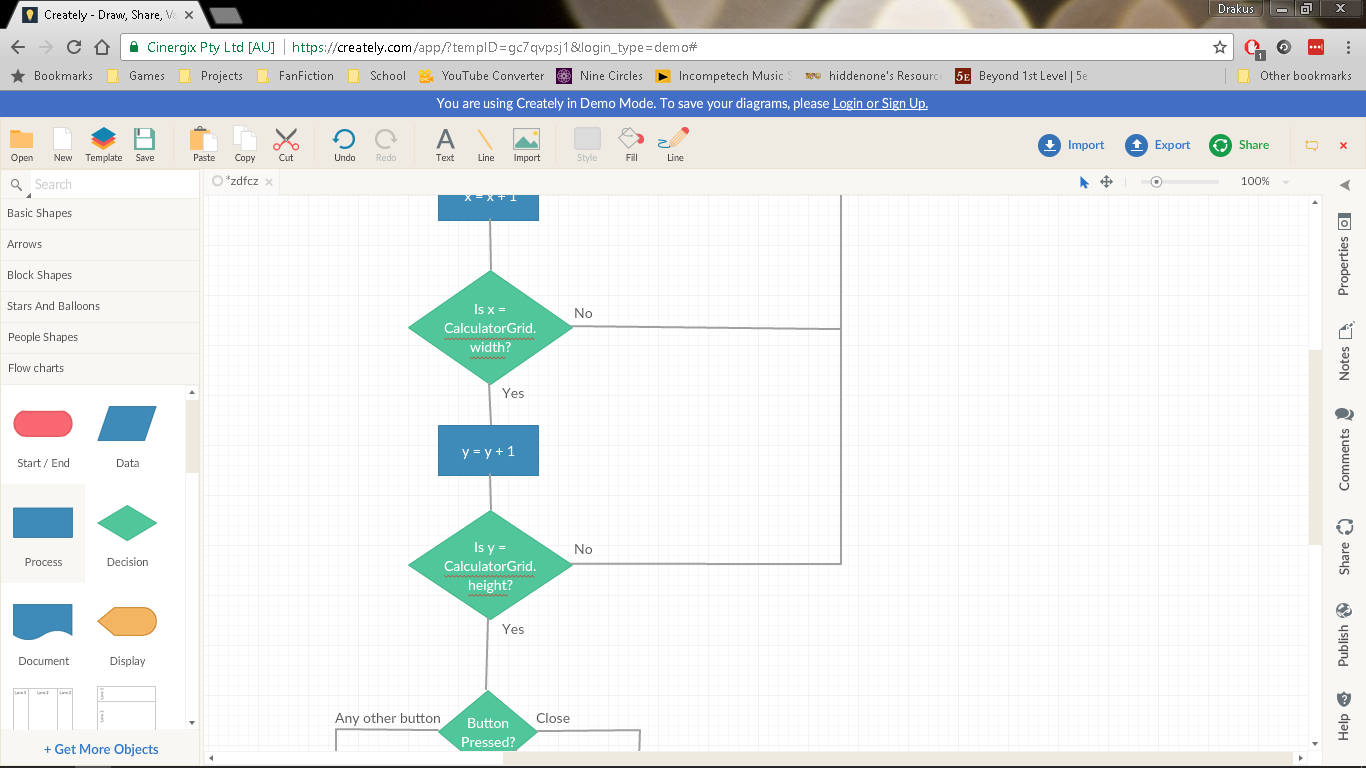
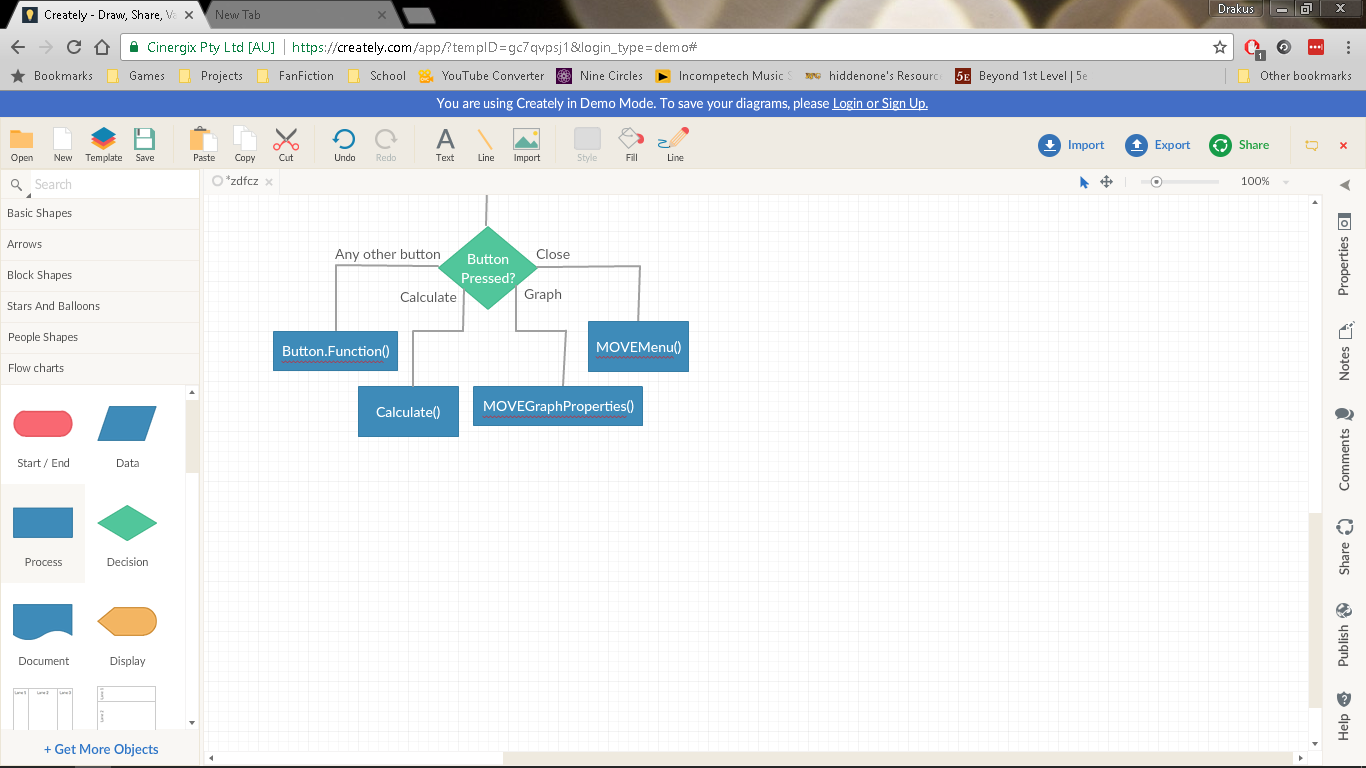
|  |  |
| --- | --- |
| Name | Function |
| Calculate | Calculates the answer to an expression given to it using the BIDMAS process. The input expression has to be in string form. |
| Differentiate | Differentiates an expression given to it using calculus. The input expression has to be in string form. |
| Menu | Creates the main menu for the program. Creates the three menu buttons at the start of the program’s runtime. |
| MOVECalculator | Moves the user to the calculator display screen. This is the screen with the various calculator buttons, including “3”, “+” and “=”, for example. |
| MOVEGraphProperties | Moves the user to the graph creation screen, which is where you can create a graph to be generated and displayed to you within specific bounds. |
| MOVEGraph | Moves the user to the graph display screen, where the graph is displayed to the user within specific bounds, and including axes if applicable to the given bounds. |



As can be seen here, the first thing that would occur when the program is started is the creation of the buttons with the labels ‘Calculator’, ‘Graph’ and ‘Exit’. These buttons are created in a vertical line as menu options. Every frame, the program would check for three things.

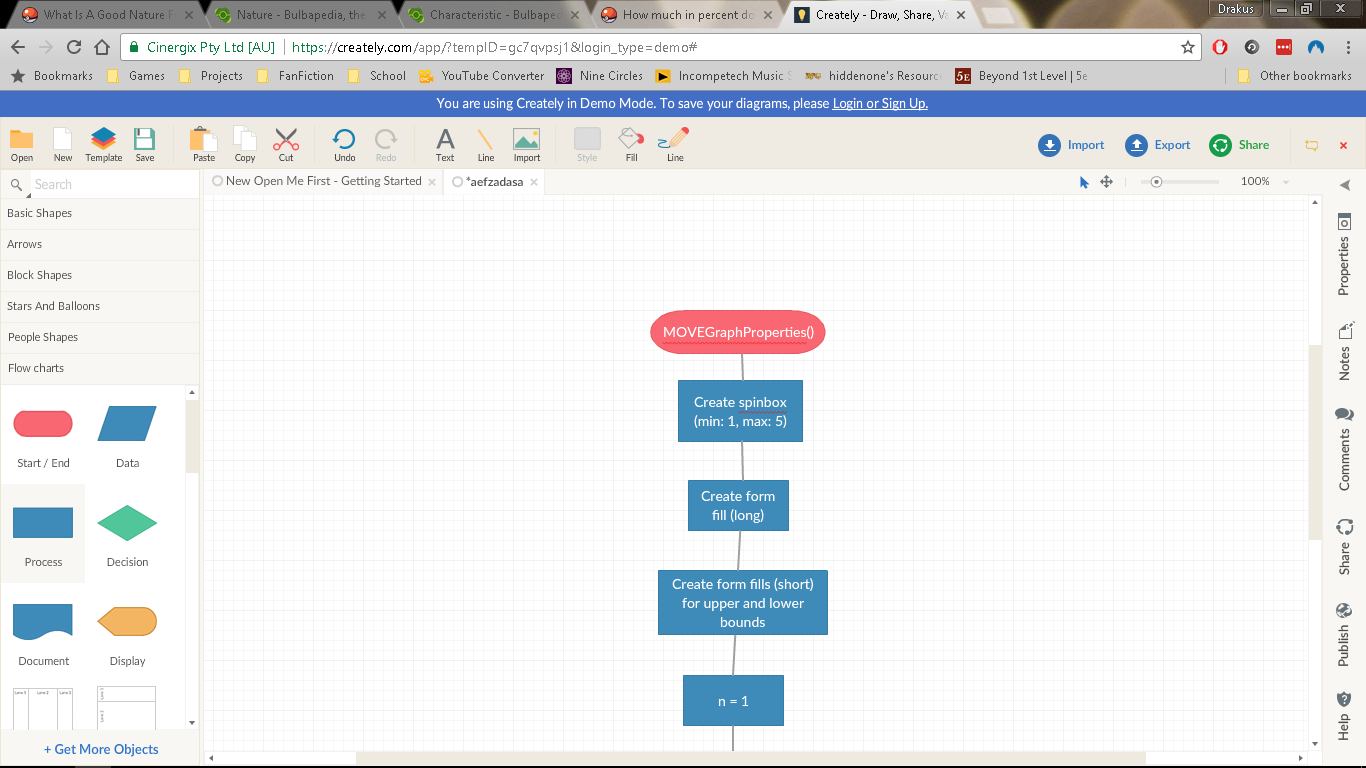
* The program would check for whether the ‘Calculator’ button has been clicked, and when it is, it sends a signal to run the MOVECalculator function which changes the screen to that of the calculator display.
* The program would check for whether the ‘Graph’ button has been clicked, and when it is, it sends a signal to run the MOVEGraphCreation function which changes the screen to that of the graph creation screen.
* The program would check for whether the ‘Exit’ button has been clicked, and when it is, it sends a signal to close the program.

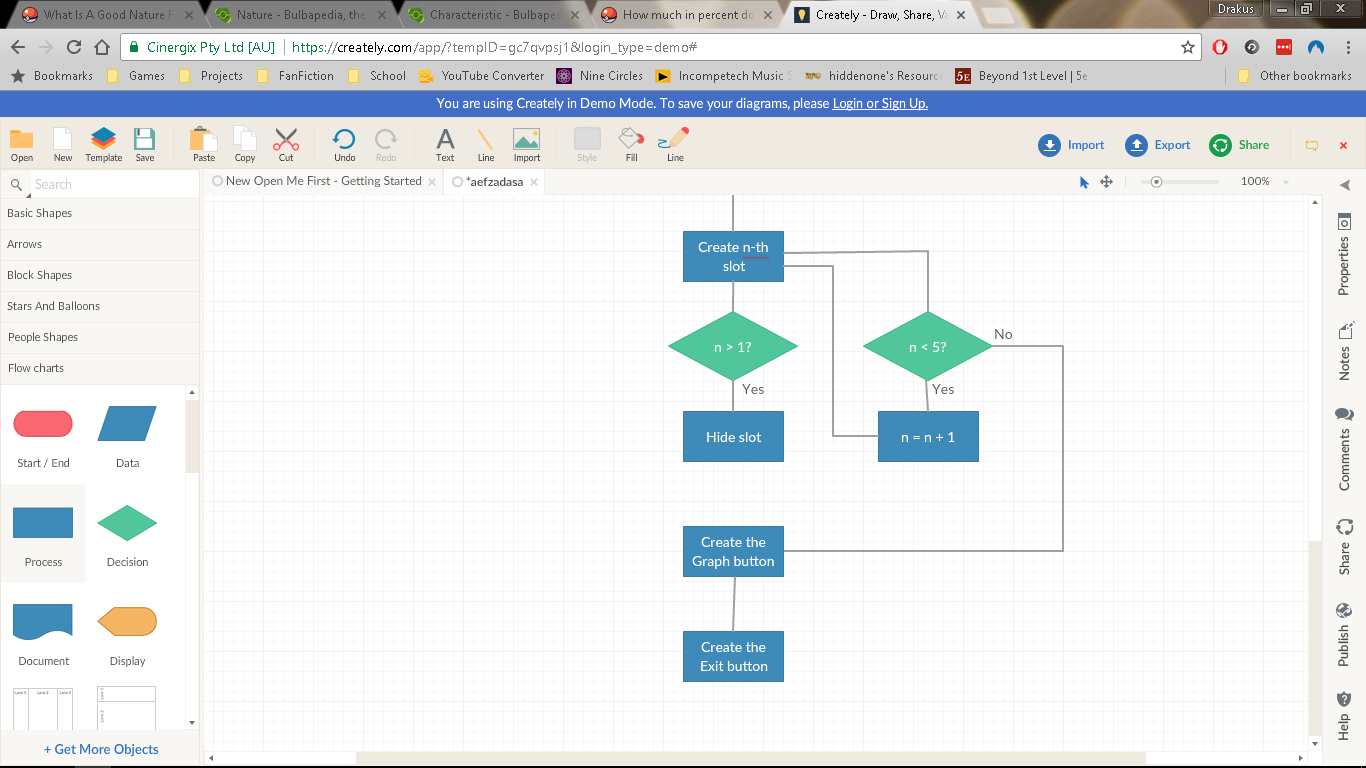




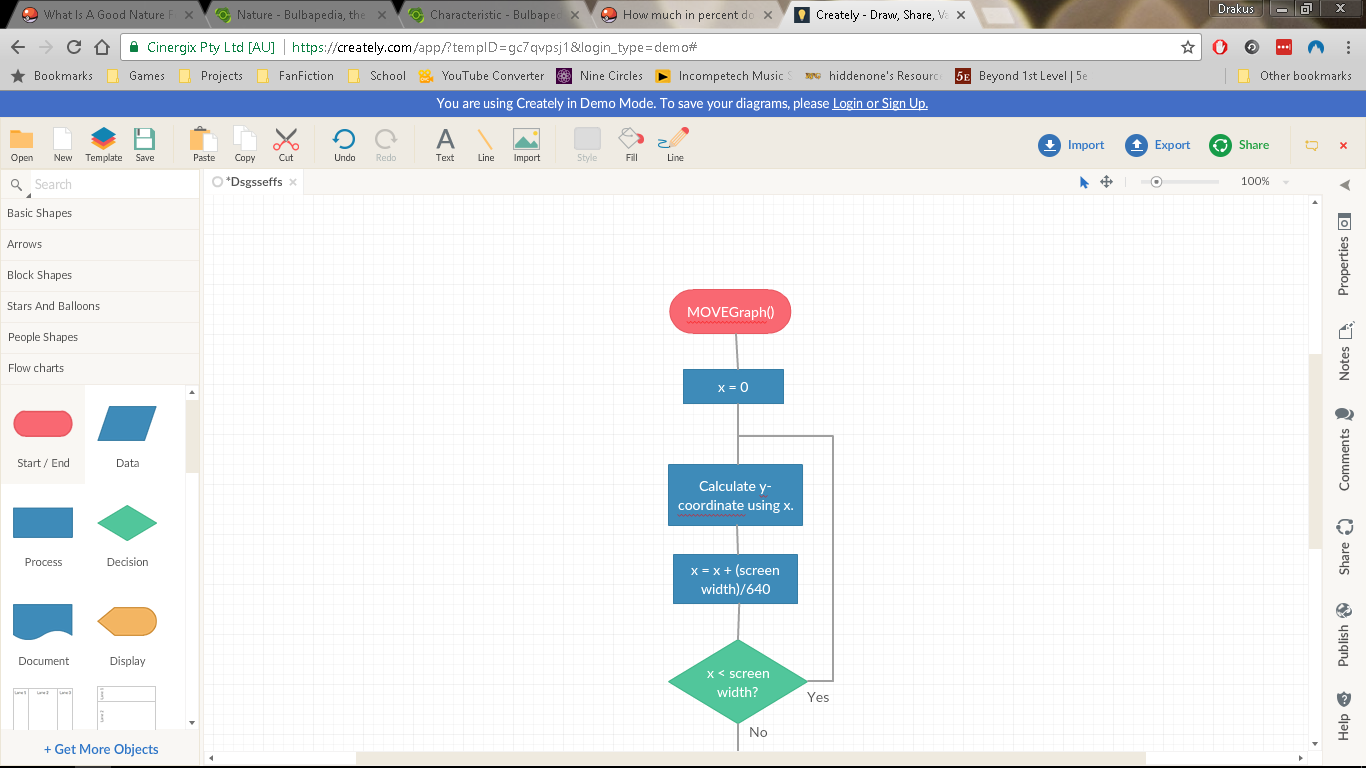
This flowchart describes the MOVECalculator function. As can be seen, the program would first read the grid which contains the calculator layout, line by line. Each index it reads the string of, it would decide what to do based on a series of selection statements.

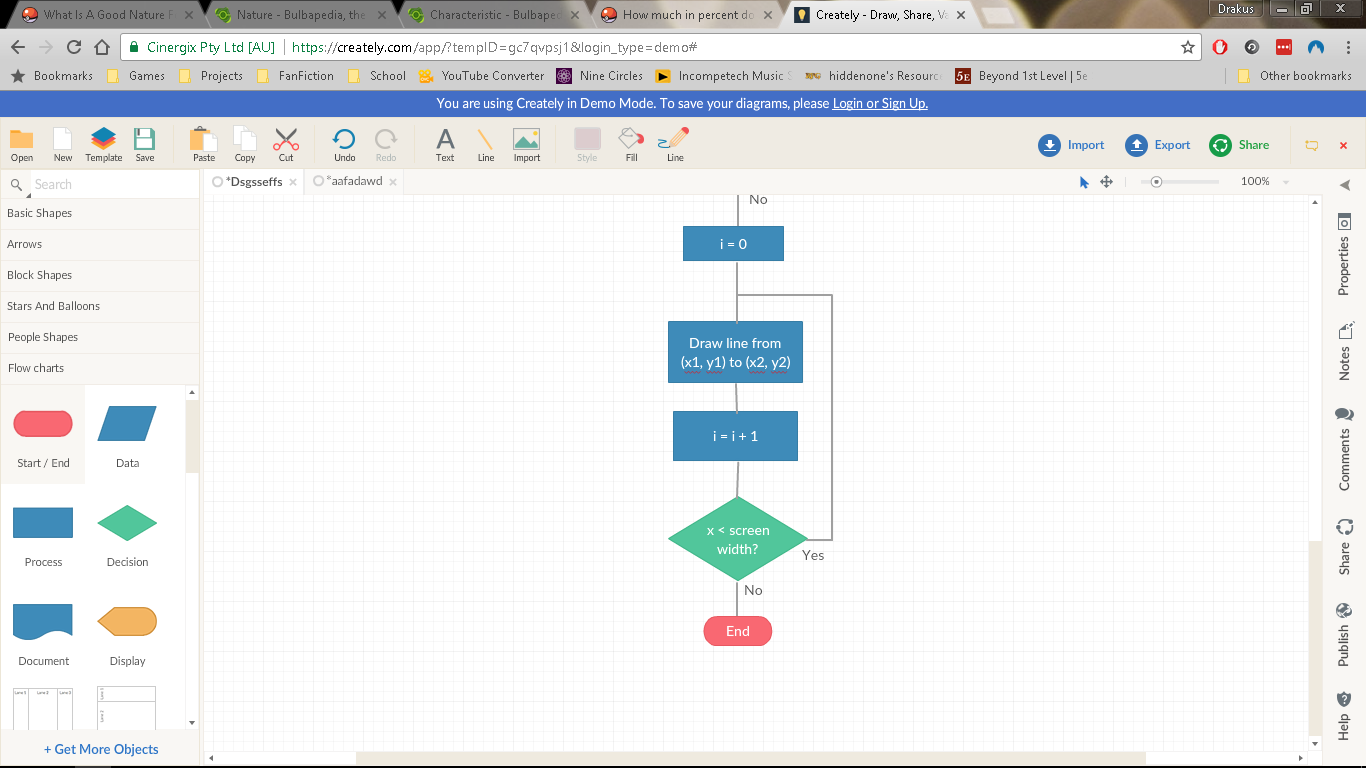
* If the grid(x, y) = “ ”, it will not create a button and will instead leave that area blank.
* If the grid(x, y) is only one character long, it will create a button of a small size.
* An index in grid(x, y) of a larger length will make the program create a button of a respective length, and will make the label the contents of that string when all spaces are eliminated. For example, an index of “ GRAPH ” would make the label become “GRAPH”.
* If an index contains “ROUND”, it will create a spin box instead of a button. This spin box will contain the number of decimal places the resulting answer is to be rounded to.





This flowchart describes the MOVEGraphProperties function. As can be seen, the program would create a spin box of a minimum value of one, and a maximum value of five. This would contain the number of curves that would be created upon the creation of the graph. The maximum limit of five was set up to prevent cluttering and also to prevent the creation process from taking too long.





This diagram describes the MOVEGraph() function. It will initiate a loop of a finite amount of iterations equal to the width of the screen in pixels. At each iteration, it calculates both the y-value of the curve at that x-value (the current number of iterations) using the Calculate function, and calculates the gradient at that x-value using the differential equation obtained before. It will store these values in lists.

After this is done, it will find where O(0, 0) (the origin of the axes of this curve) is in the window, and draw a horizontal line at its y-position and a vertical line at its x-position. These become the axes of the curve, which are relative in their placement on the screen based on the position of the curve in relation to them.

After the axes are drawn, it will again iterate like before, drawing a line from the point on the screen given by (i, Points[i]) to the point given by (i+1, Points[i+1]). This means that the curve will be drawn assuming a perfect area obtained by the trapezium rule, with the number of trapeziums used being . This means that if zoomed into far enough, the curve would be very angular, not actually being a curve at all but a collection of joined up straight lines with varying angles. However, this lack of resolution shouldn’t matter due to the curve creating enough trapeziums to make this effect almost impossible to distinguish to the human user.

After the graph is created and drawn, it creates a ‘Close’ button in the top left corner. This ‘Close’ button calls the MOVEMenu() procedure to recall the menu screen.

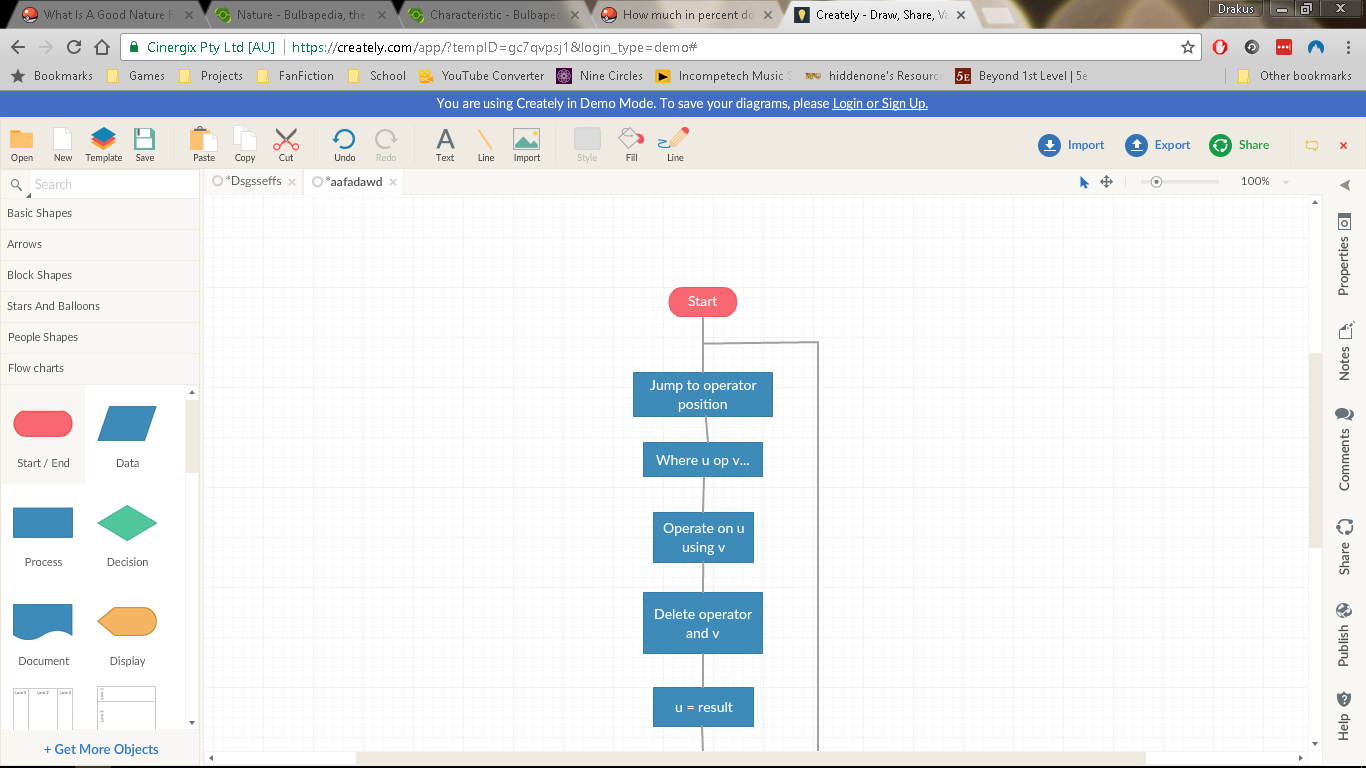
This flowchart describes the Calculate() function. As can be seen, the program will take the expression given to it in a string format, splitting the string into a list. After this, the program will convert each numerical character into an integer and will combine them into a more complex integer or float.

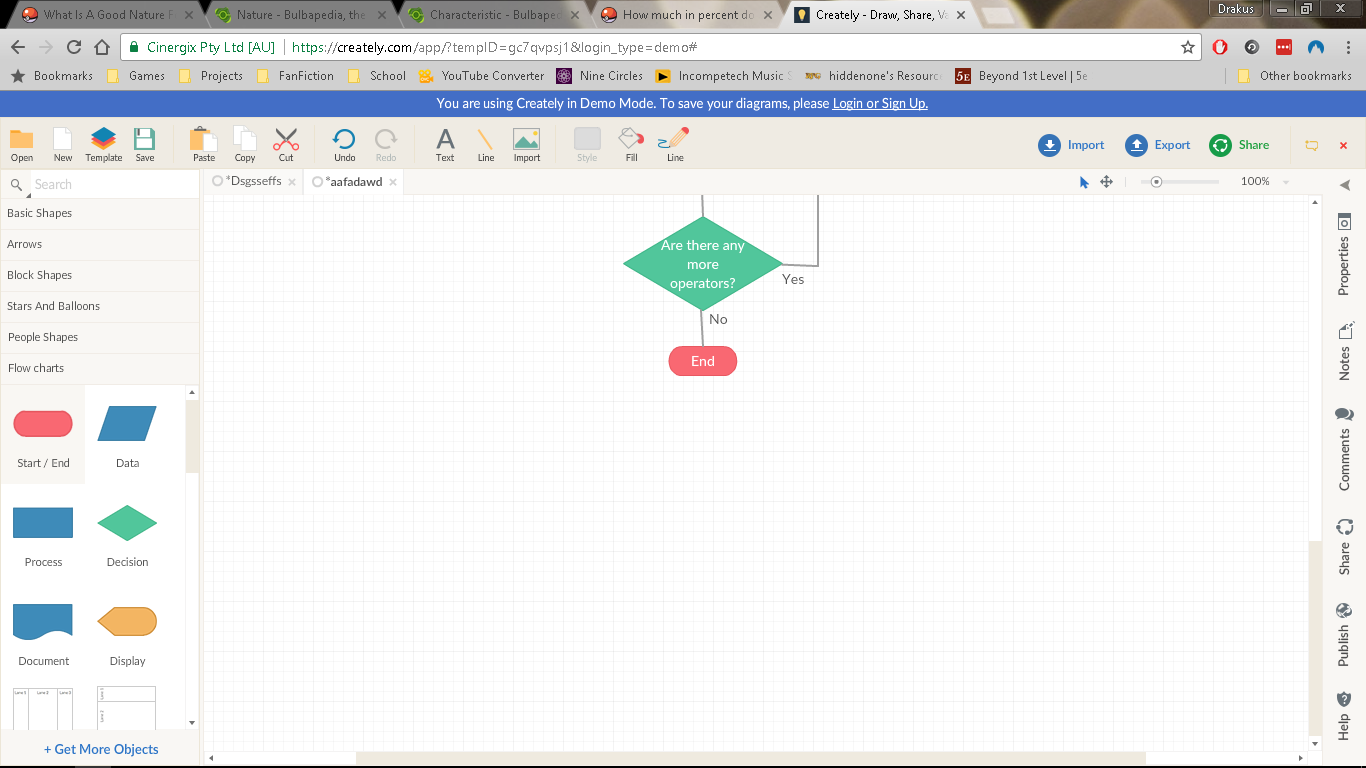
Then, it would search the list for variables as defined by the list that are passed into the function by the calling script, and substitutes them for the float or integer values they represent.

After the preparation is done, the program would determine whether the expression given is to be calculated upon or differentiated, and thus calls the respective function (either Calculate() or Differentiate()) accordingly.

Inside the Calculate function, a function called Functions() is run. The program would iterate through the functions, calling the function at that particular index on the number directly after the letter signpost. If there are no letters corresponding to a particular function, it skips it.

After all the functions have been accounted for, it would go through the BIDMAS operations, checking if there are any operators tied to that function in the newly split list. If there are none, it moves on to the next of the BIDMAS operations, however if there are one or more, it calls the respective function and calculates the result inside of it, substituting it back into the list.





Each of the calculation functions except Brackets() work largely similarly to the way that Functions() works, with the program jumping to the index of the first instance of the operator in the list and recording the values before and after it. After this is done, the necessary operation would be performed on the values recorded with the resulting value substituted back into the list at the place of the operators. The two adjacent values would then be deleted from the list. This would be repeated with each successive operator as that character becomes the new first operator of that type in the list.

Brackets() would work by finding the first instance of the open bracket character in the list, and recording its position. Then, it would continue through the list, updating the open bracket position if it encounters any more. Upon reaching a closed bracket character, it records its position and calls Calculate() on the joined characters between them, replacing all characters in the list between the open bracket and closed bracket, inclusive, with the result. This will be repeated until there are no more open brackets in the list. If the number of open and closed brackets is unequal, the program will skip the iteration and will just output a Parse Error.