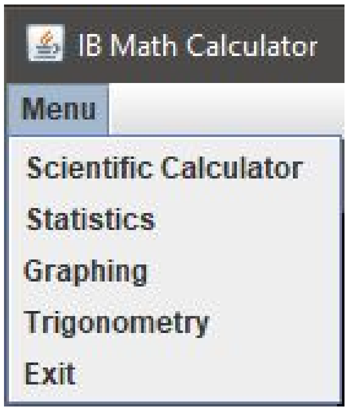
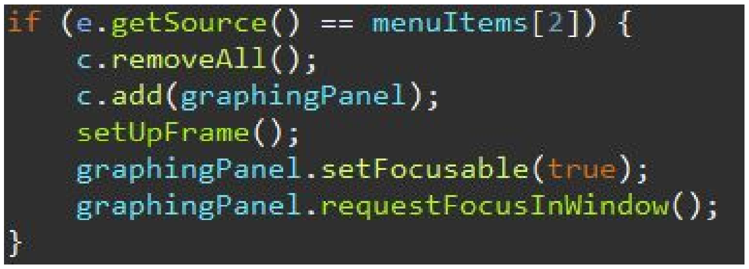
Criterion C: Development

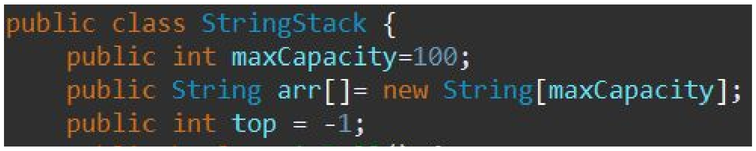
This program is written in java. This program is a calculator for IB HL Math Students. Its functionalities consist of a scientific calculator, a statistics mode, a graphing calculator, and a trigonometry mode.

JFrame	
Explanation	**Image**
There is one frame that contains a menubar, menu, and menu items. This JFrame extension contains a container that allows for different panels to be placed on the JFrame. This is the highest level in the program structure as it is always present. Only one JFrame was used to ensure the maintenance of code was minimal and organized. Additionally, the JPanels were placed in the container as opposed to JFrames since JFrames have a graphical stutter. To the right is an image of the menu buttons.	 IB Math Calculator Menu Scientific Calculator Statistics Graphing Trigonometry Exit

To the right is the code for changing the JPanel the container contains depending on the user clicking the menu items. For example, if the user clicks "Graphing" in the menu, **menuItems[2]** is clicked and the graphing calculator JPanel is displayed.	 `if (e.getSource() == menuItems[2]) {`
	`    c.removeAll();`
	`    c.add(graphingPanel);`
	`    setUpFrame();`
	`    graphingPanel.setFocusable(true);`
	`    graphingPanel.requestFocusInWindow();`
	`}`

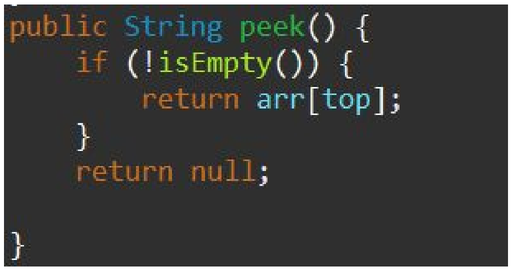
StringStack and DoubleStack

A class for a stack of string elements implemented from scratch was called StringStack. The stack was coded as an array and had methods that were specific to a stack.



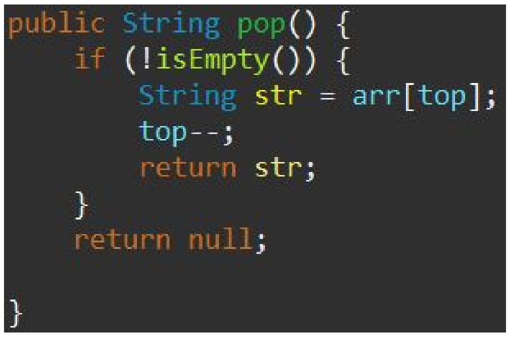
```java
public class StringStack {
    public int maxCapacity=100;
    public String arr[]= new String[maxCapacity];
    public int top = -1;
```

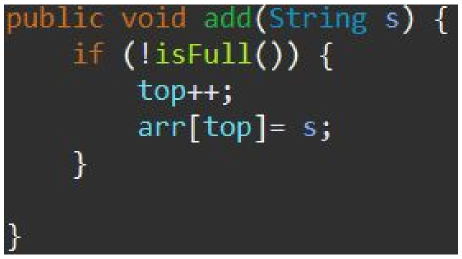
There is a peek() method that returns the top element of the stack without removing it from the stack.

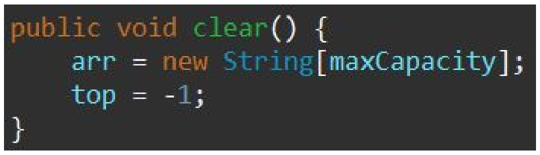


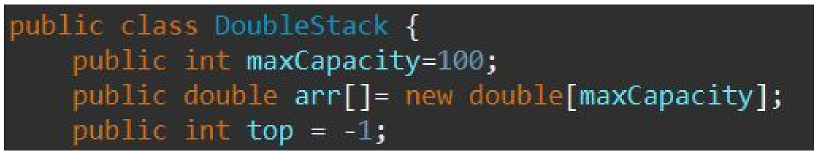
```java
public String peek() {
    if (!isEmpty()) {
        return arr[top];
    }
    return null;

}
```

There is a pop() method that removes and returns the top element on the stack.


```java
public String pop() {
    if (!isEmpty()) {
        String str = arr[top];
        top--;
        return str;
    }
    return null;

}
```
Additionally, there is a add() method that adds a string as an element to the stack.

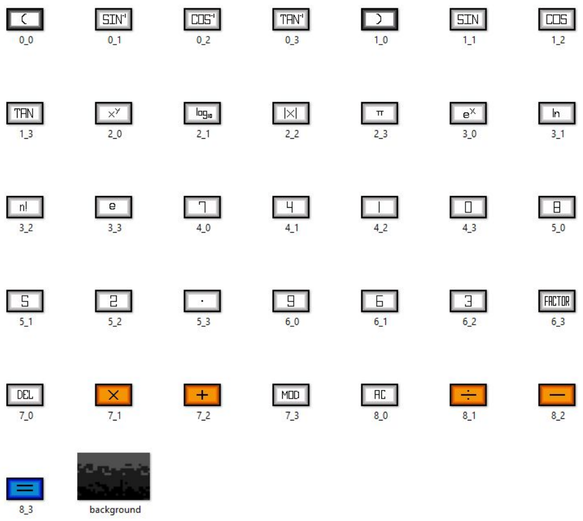

```java
public void add(String s) {
    if (!isFull()) {
        top++;
        arr[top]= s;
    }

}
```
Also, the clear() method removes all elements within the stack.


```java
public void clear() {
    arr = new String[maxCapacity];
    top = -1;
}
```
Similarly, a class for a stack of double elements was also created from scratch and was called DoubleStack. DoubleStack also has all the methods that StringStack has. The use of stacks will later prove useful. Below is the code for the class DoubleStack.

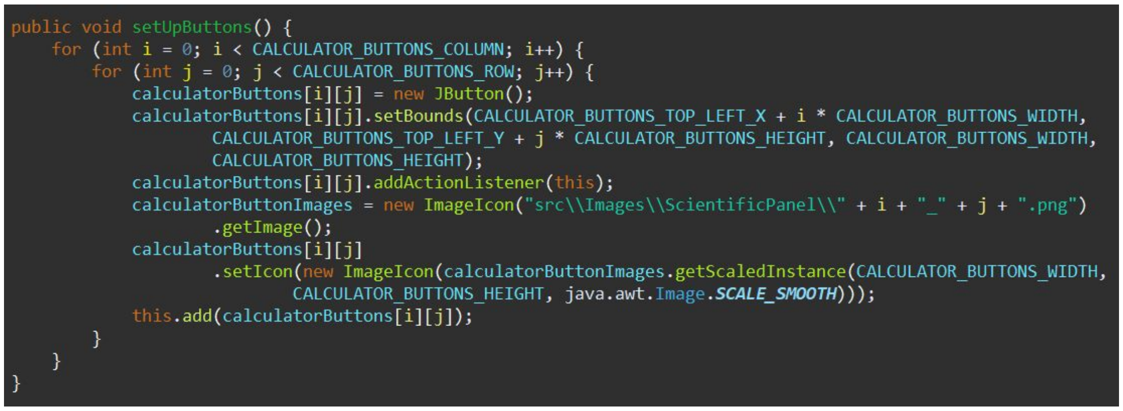

```java
public class DoubleStack {
    public int maxCapacity=100;
    public double arr[]= new double[maxCapacity];
    public int top = -1;
```

Scientific Panel

The **ScientificPanel** is a JPanel that allows the user to evaluate math expressions.
The GUI used for this JPanel consists of JButtons, JLabels, paint(), and Images that reference photos within the source folder as shown to the right.

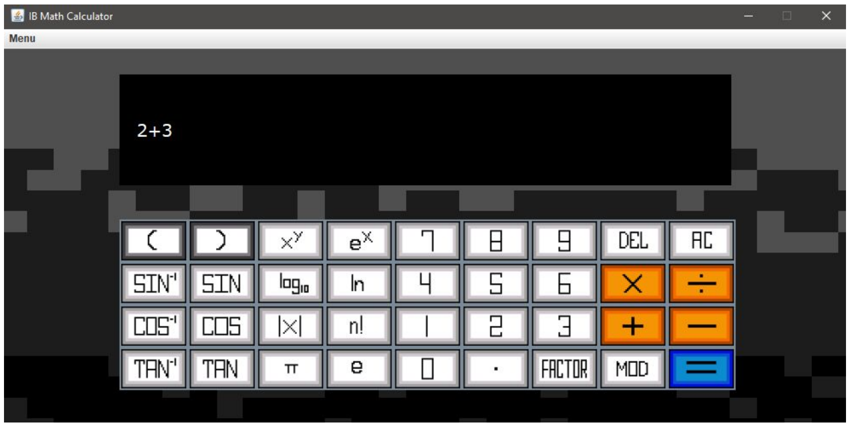


There is a 2-D array of JButtons which is appropriate as the buttons are laid out in a 9 by 4 grid. Each button is directed to a specific position, has ActionListener added, has an image and is added to the JPanel. The naming convention for the images corresponds to files in the folder.

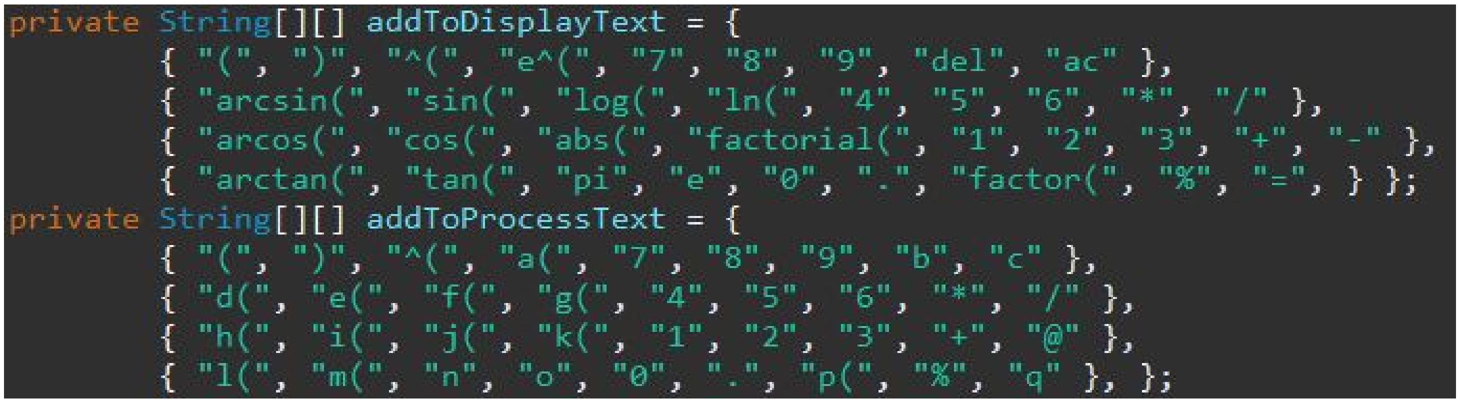


```java
public void setUpButtons() {
    for (int i = 0; i < CALCULATOR_BUTTONS_COLUMN; i++) {
        for (int j = 0; j < CALCULATOR_BUTTONS_ROW; j++) {
            calculatorButtons[i][j] = new JButton();
            calculatorButtons[i][j].setBounds(CALCULATOR_BUTTONS_TOP_LEFT_X + i * CALCULATOR_BUTTONS_WIDTH,
                    CALCULATOR_BUTTONS_TOP_LEFT_Y + j * CALCULATOR_BUTTONS_HEIGHT, CALCULATOR_BUTTONS_WIDTH,
                    CALCULATOR_BUTTONS_HEIGHT);
            calculatorButtons[i][j].addActionListener(this);
            calculatorButtonImages = new ImageIcon("src\\Images\\ScientificPanel\\" + i + "_" + j + ".png")
                    .getImage();
            calculatorButtons[i][j]
                    .setIcon(new ImageIcon(calculatorButtonImages.getScaledInstance(CALCULATOR_BUTTONS_WIDTH,
                            CALCULATOR_BUTTONS_HEIGHT, java.awt.Image.SCALE_SMOOTH)));
            this.add(calculatorButtons[i][j]);
        }
    }
}
```

The user can click buttons to enter operators and operands into the black JTextField which will be evaluated when the user clicks the “=” JButton. For example, if the user clicks the JButtons “2”, then “+”, then “3”, the display will be as shown below.

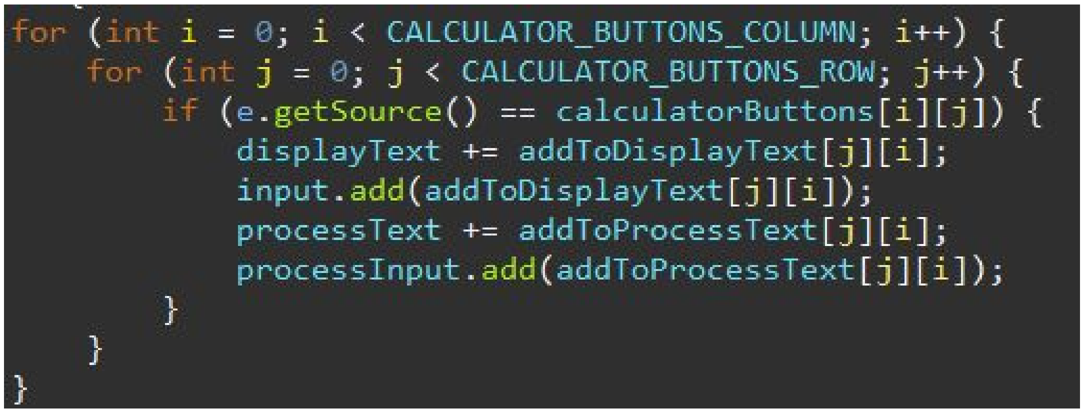

IB Math Calculator
Menu
2+3
( ) x^y e^x 7 8 9 DEL AC
SIN^-1 SIN log10 ln 4 5 6 × ÷
COS^-1 COS |x| n! 1 2 3 + −
TAN^-1 TAN π e 0 . FACTOR MOD =

To display this math expression, there are two strings, **displayText** and **processText**. Both strings keep track of the operators and operands added and represent the same information. However, the notation used to represent the information is different for the two strings. This is because the **eval()** method requires a specific string input which will be further explained later on. The **displayText** is the string that is shown to the user as they click buttons, but the **processText** string is what the program uses to evaluate the math expression and is never shown to the user. Additionally, there are two stacks called **input** and **processInput** which contain string elements. These stacks both also keep track of the operators and operands added and again both represent the same information, but with different notation. The reason a stack is used is because of the “del” button which will be further explained later on. Each button has a corresponding text that will be added to the **displayText** and **processText** when it is clicked. Below is an array of the text that is added to the **displayText** string and **processText** string respectively.



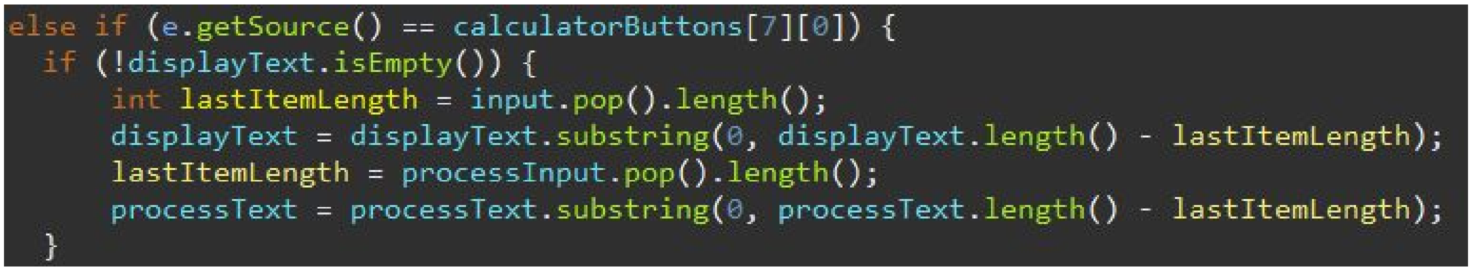
```java
private String[][] addToDisplayText = {
        { "(", ")", "^(", "e^(", "7", "8", "9", "del", "ac" },
        { "arcsin(", "sin(", "log(", "ln(", "4", "5", "6", "*", "/" },
        { "arcos(", "cos(", "abs(", "factorial(", "1", "2", "3", "+", "-" },
        { "arctan(", "tan(", "pi", "e", "0", ".", "factor(", "%", "=", } };
private String[][] addToProcessText = {
        { "(", ")", "^(", "a(", "7", "8", "9", "b", "c" },
        { "d(", "e(", "f(", "g(", "4", "5", "6", "*", "/" },
        { "h(", "i(", "j(", "k(", "1", "2", "3", "+", "@" },
        { "l(", "m(", "n", "o", "0", ".", "p(", "%", "q" }, };
```

For example, if the user clicks the “sin(” button, the corresponding text that will be added to the displayText is “sin(” and the corresponding text that is added to the processText is “e(”. Similarly, “sin(” and “e(” are added to the input and processInput stacks as well. Below is the code in the ActionPerformed() method.



```
for (int i = 0; i < CALCULATOR_BUTTONS_COLUMN; i++) {
    for (int j = 0; j < CALCULATOR_BUTTONS_ROW; j++) {
        if (e.getSource() == calculatorButtons[i][j]) {
            displayText += addToDisplayText[j][i];
            input.add(addToDisplayText[j][i]);
            processText += addToProcessText[j][i];
            processInput.add(addToProcessText[j][i]);
        }
    }
}
```

There is an option for the user to delete the last operator/operand added in the math string expression by clicking the button “del”. Thus, the stack data type of **input** and **processInput** is very effective as the last operator/operand added can be popped off the stack. The operators/operands elements are of different lengths so using a string would make the process of removing the last operator/operand very cumbersome, while stacks are very effective for this process. Also, stacks allow the programmer to store data in a “first in last out” procedure which is very effective as the last/next operator/operand is being added/removed. Below is the code for the “del” button in the **ActionPerformed()** method.



```
else if (e.getSource() == calculatorButtons[7][0]) {
  if (!displayText.isEmpty()) {
      int lastItemLength = input.pop().length();
      displayText = displayText.substring(0, displayText.length() - lastItemLength);
      lastItemLength = processInput.pop().length();
      processText = processText.substring(0, processText.length() - lastItemLength);
  }
```

eval()

When the user clicks the “=” button, the **eval()** method takes in the user-entered math string expression and evaluates it. The string must have a special notation such as “@” instead of “-” which is why the **processText** string is used. This method has two stacks, one for the operators and one for the operands. The operators are contained within the **StringStack operator**. The operands are contained within the **DoubleStack term**. Stacks are very effective for this method since operators and operands have priorities and operators and terms can easily be added/removed from the top of the stack despite having different lengths.

<u>Algorithm for eval()</u>

The parameter text is looped through from left to right and as operators/operands appear, they are added to their corresponding stacks. If an operator that is being added has a lower priority than the previous top element of the stack, then the previous element is removed from the stack and evaluates it with the corresponding operands.

Ex: The parameter string is “23+3*4+5+6”

Loop and find an operand which is “23”.
Add “23” to **term** stack
operator stack from bottom to top: []
term stack from bottom to top: [23]

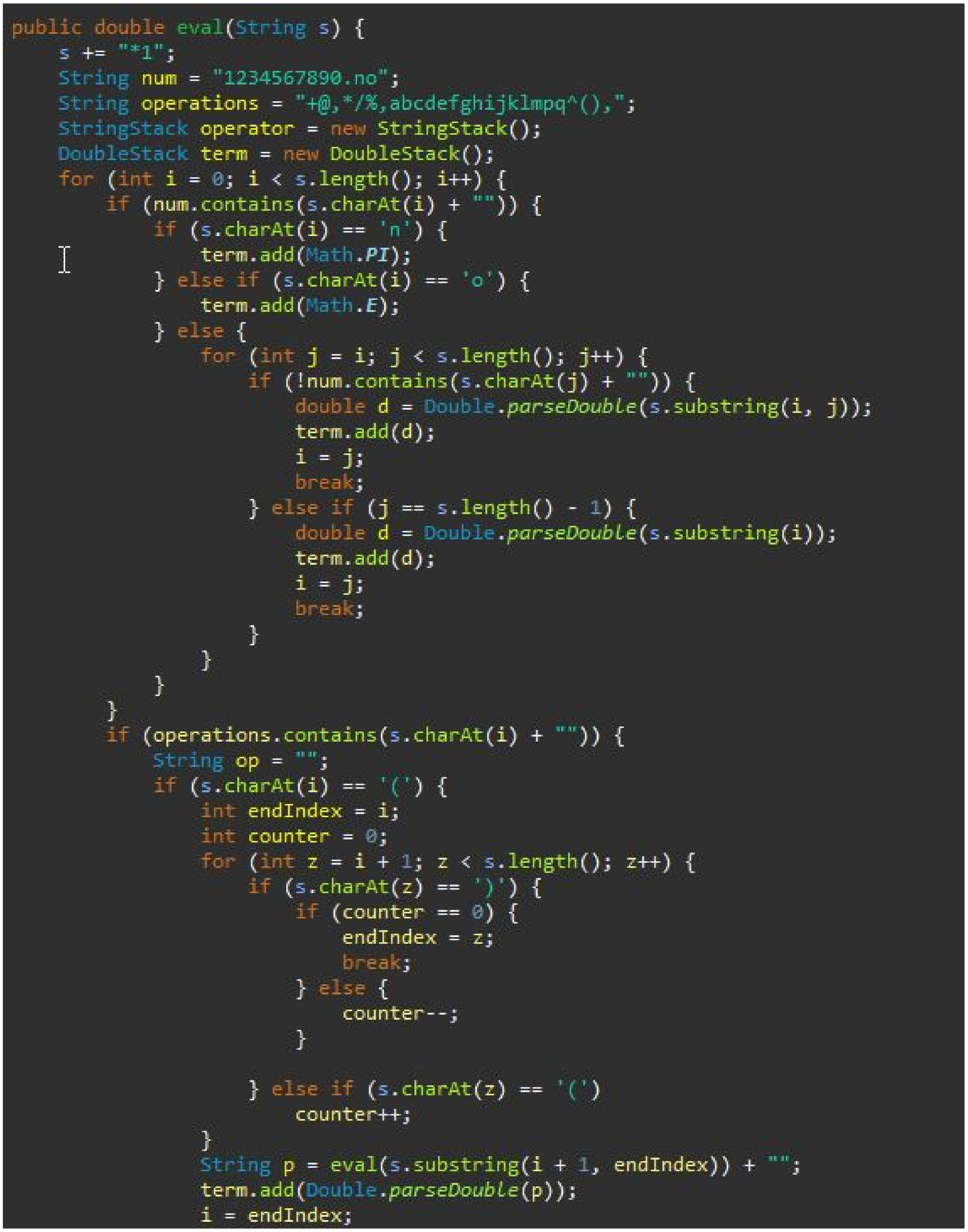
Loop and find an operator which is “+”.
Add “+” to **operator** stack
operator stack from bottom to top: [+]
term stack from bottom to top: [23]

Loop and find an operand which is “3”.
Add “3” to **term** stack
operator stack from bottom to top: [+]
term stack from bottom to top: [23,3]

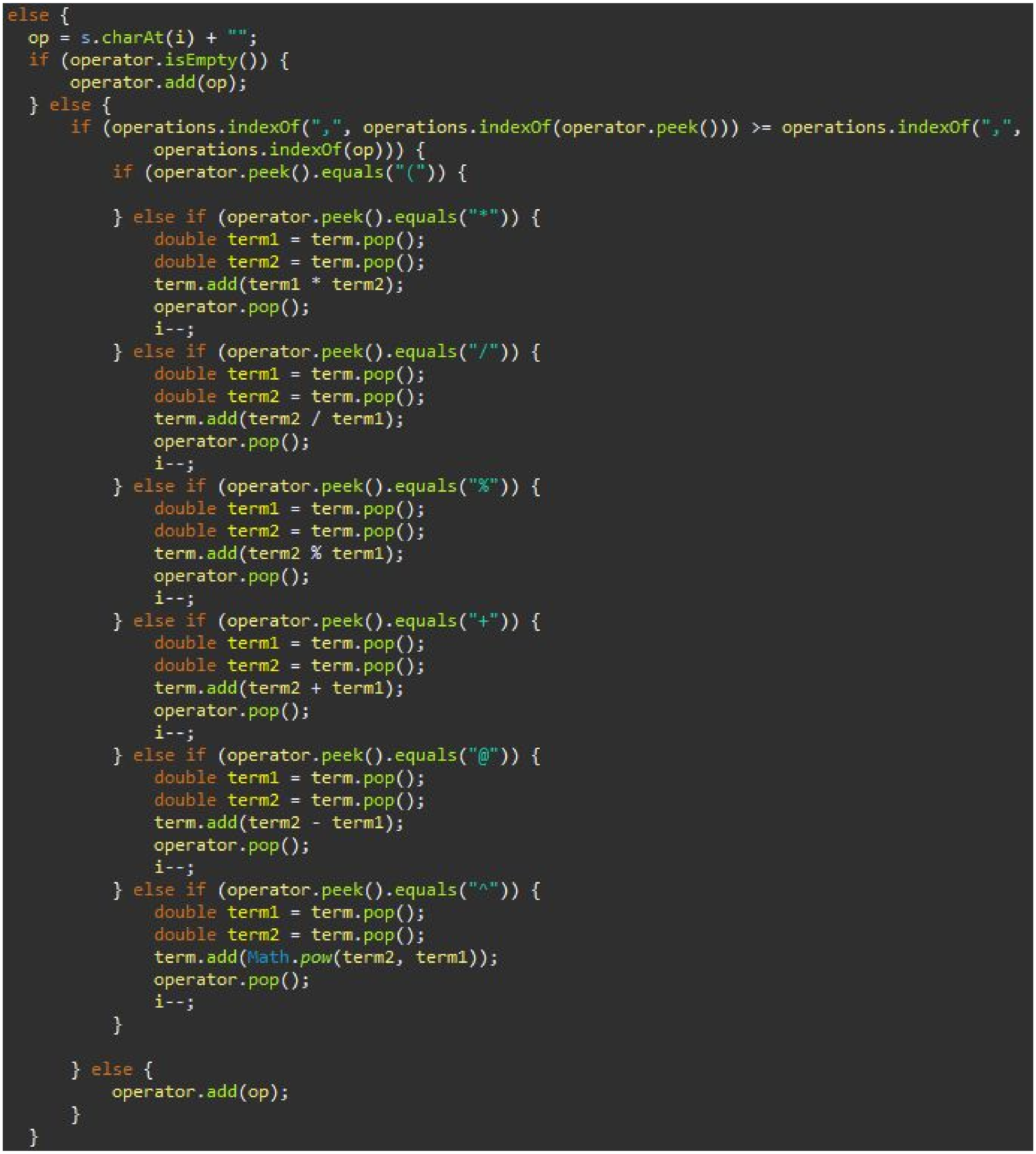
Loop and find an operator which is “*”.
Since “*” has a higher priority than the top element of the **term** stack which is “+”, add “*” to **term** stack without evaluating anything.
operator stack from bottom to top: [+,*]
term stack from bottom to top: [23,3]

Loop and find an operand which is “4”.
Add “4” to **term** stack
operator stack from bottom to top: [+,*]
term stack from bottom to top: [23,3,4]
Loop and find an operator which is “+”.
Since “+” has a lower priority than the top element of the **term** stack which is “*”, evaluate the “*” operator with the top two elements of the **term** stack which are “3” and “4”, giving a result of “12”. Then, remove the “*” element from the **operator** stack and add the “+” to the **operator** stack and add the “12” to the **term** stack.
operator stack from bottom to top: [+,+]
term stack from bottom to top: [23,12]

This algorithm continues until the math string expression is evaluated. If there are brackets within the string, the expression inside the brackets must first be evaluated. Thus, recursion is needed as the function must call itself to evaluate the math string expression inside the brackets. Additionally, the use of recursion over a loop provides a code that requires low maintenance, is well organized and is clear. Below is the recursive code for the brackets.

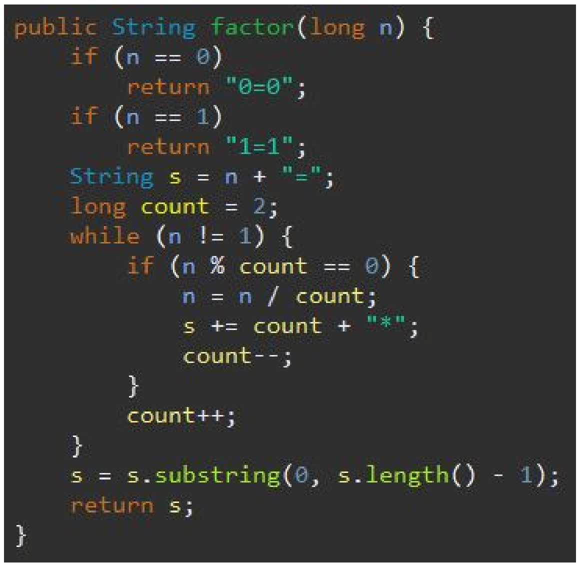


```java
public double eval(String s) {
    s += "*1";
    String num = "1234567890.no";
    String operations = "+@,*/%,abcdefghijklmpq^(),";
    StringStack operator = new StringStack();
    DoubleStack term = new DoubleStack();
    for (int i = 0; i < s.length(); i++) {
        if (num.contains(s.charAt(i) + "")) {
            if (s.charAt(i) == 'n') {
                term.add(Math.PI);
            } else if (s.charAt(i) == 'o') {
                term.add(Math.E);
            } else {
                for (int j = i; j < s.length(); j++) {
                    if (!num.contains(s.charAt(j) + "")) {
                        double d = Double.parseDouble(s.substring(i, j));
                        term.add(d);
                        i = j;
                        break;
                    } else if (j == s.length() - 1) {
                        double d = Double.parseDouble(s.substring(i));
                        term.add(d);
                        i = j;
                        break;
                    }
                }
            }
        }
        if (operations.contains(s.charAt(i) + "")) {
            String op = "";
            if (s.charAt(i) == '(') {
                int endIndex = i;
                int counter = 0;
                for (int z = i + 1; z < s.length(); z++) {
                    if (s.charAt(z) == ')') {
                        if (counter == 0) {
                            endIndex = z;
                            break;
                        } else {
                            counter--;
                        }

                    } else if (s.charAt(z) == '(')
                        counter++;
                }
                String p = eval(s.substring(i + 1, endIndex)) + "";
                term.add(Double.parseDouble(p));
                i = endIndex;
```

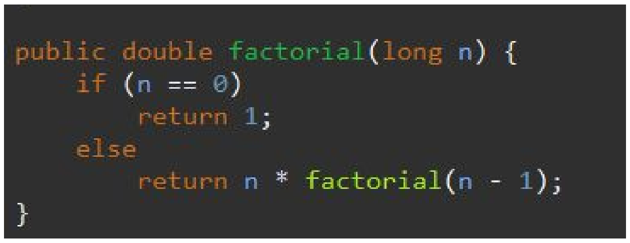


```java
else {
  op = s.charAt(i) + "";
  if (operator.isEmpty()) {
      operator.add(op);
  } else {
      if (operations.indexOf(",", operations.indexOf(operator.peek())) >= operations.indexOf(",",
              operations.indexOf(op))) {
          if (operator.peek().equals("(")) {

          } else if (operator.peek().equals("*")) {
              double term1 = term.pop();
              double term2 = term.pop();
              term.add(term1 * term2);
              operator.pop();
              i--;
          } else if (operator.peek().equals("/")) {
              double term1 = term.pop();
              double term2 = term.pop();
              term.add(term2 / term1);
              operator.pop();
              i--;
          } else if (operator.peek().equals("%")) {
              double term1 = term.pop();
              double term2 = term.pop();
              term.add(term2 % term1);
              operator.pop();
              i--;
          } else if (operator.peek().equals("+")) {
              double term1 = term.pop();
              double term2 = term.pop();
              term.add(term2 + term1);
              operator.pop();
              i--;
          } else if (operator.peek().equals("@")) {
              double term1 = term.pop();
              double term2 = term.pop();
              term.add(term2 - term1);
              operator.pop();
              i--;
          } else if (operator.peek().equals("^")) {
              double term1 = term.pop();
              double term2 = term.pop();
              term.add(Math.pow(term2, term1));
              operator.pop();
              i--;
          }

      } else {
          operator.add(op);
      }
  }
```

Also, there are **factor**() and **factorial**() methods as shown below which are used in the **eval**() method.



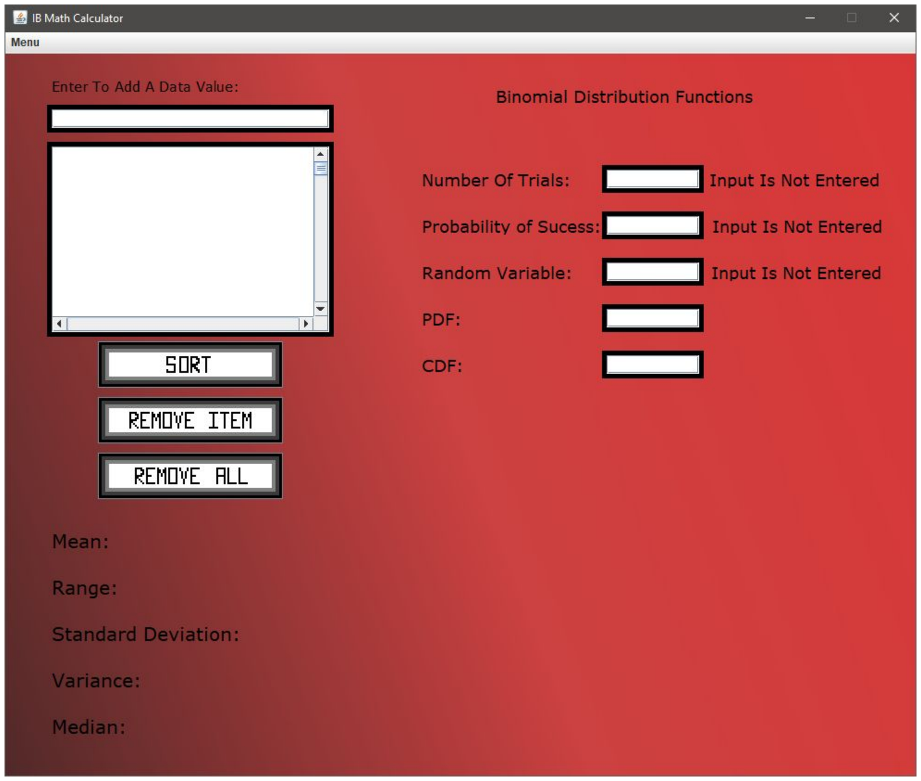
```java
public String factor(long n) {
    if (n == 0)
        return "0=0";
    if (n == 1)
        return "1=1";
    String s = n + "=";
    long count = 2;
    while (n != 1) {
        if (n % count == 0) {
            n = n / count;
            s += count + "*";
            count--;
        }
        count++;
    }
    s = s.substring(0, s.length() - 1);
    return s;
}
```



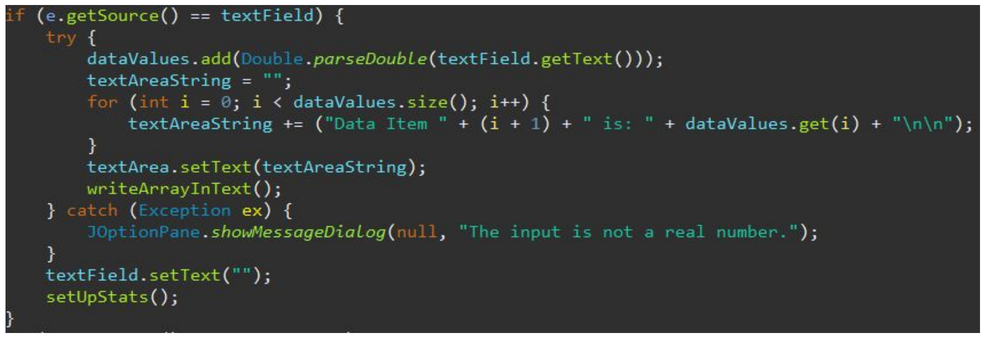
```java
public double factorial(long n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

StatisticsPanel

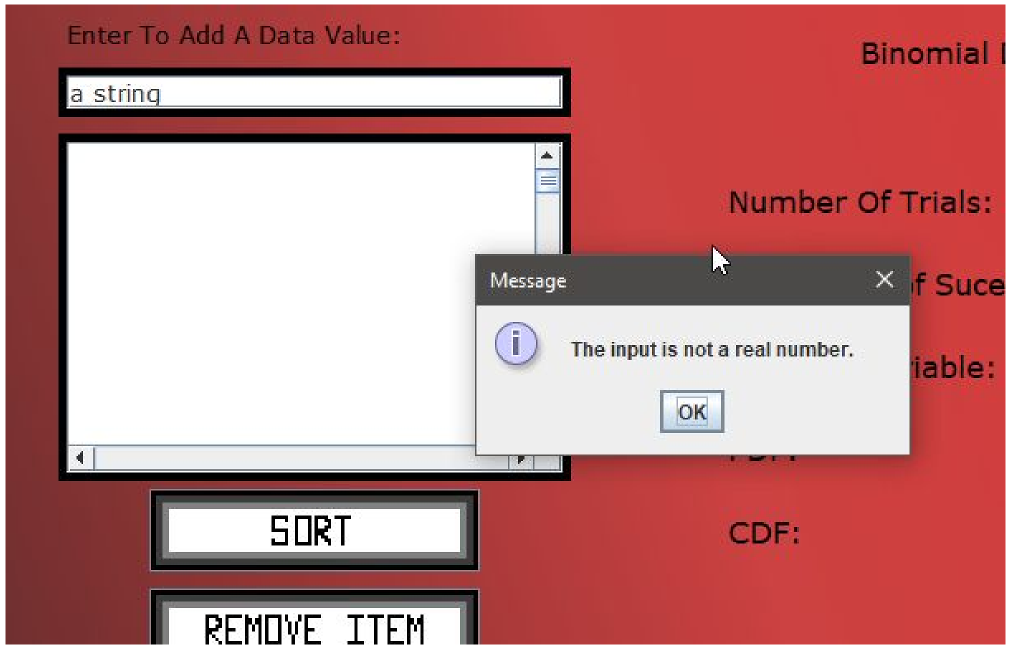
The **StatisticsPanel** is a JPanel that allows the user to evaluate statistical functions for given data.

Explanation	**Image**
GUI such as JTextField, JButton, JLabel, JTextArea, JScrollPane, and Image were used to create the panel. JTextfields are appropriate as it easily allows the user to input values with the keyboard. The JButtons allow for easy functionality as only a mouse click is needed. The JTextArea provides a display of the values the user input and the JScrollPane helps the user view data that may not fit in the display.	 IB Math Calculator Menu Enter To Add A Data Value: SORT REMOVE ITEM REMOVE ALL Mean: Range: Standard Deviation: Variance: Median: Binomial Distribution Functions Number Of Trials: Input Is Not Entered Probability of Sucess: Input Is Not Entered Random Variable: Input Is Not Entered PDF: CDF:

The user inputs in the text fields are done within a **try** and **catch** to ensure the input is valid. For example, when the user enters a data value, it has to be a real number. Otherwise, an error message will be displayed such as when “a string” is inputted as shown to the right with its code.

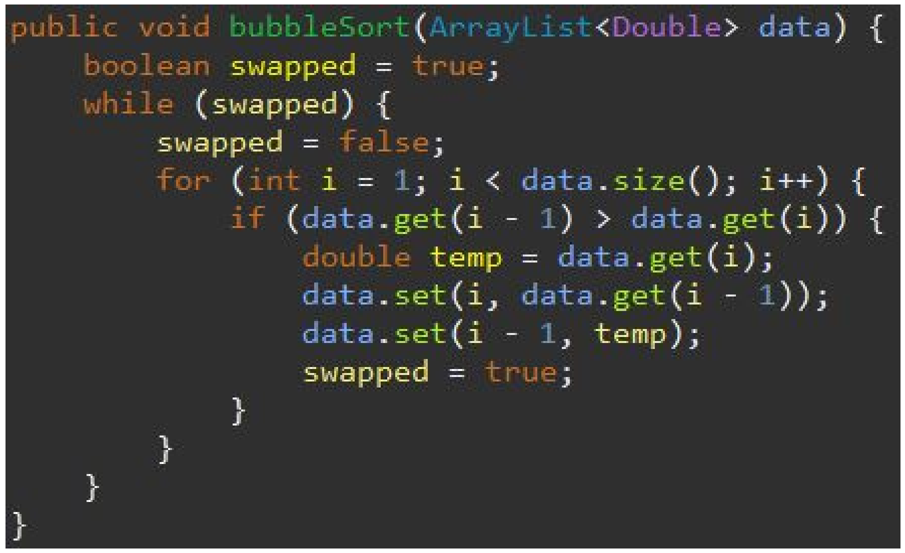


```java
if (e.getSource() == textField) {
    try {
        dataValues.add(Double.parseDouble(textField.getText()));
        textAreaString = "";
        for (int i = 0; i < dataValues.size(); i++) {
            textAreaString += ("Data Item " + (i + 1) + " is: " + dataValues.get(i) + "\n\n");
        }
        textArea.setText(textAreaString);
        writeArrayInText();
    } catch (Exception ex) {
        JOptionPane.showMessageDialog(null, "The input is not a real number.");
    }
    textField.setText("");
    setUpStats();
}
```



The input is not a real number.

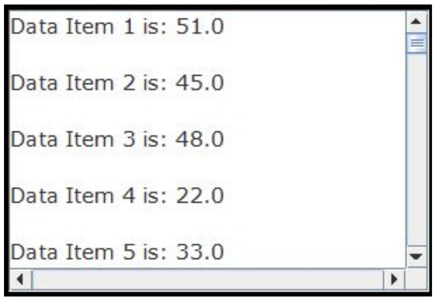
The inputted data from the user is added to an **ArrayList** of type double. As the three buttons on JPanel describe, the buttons sort the values, remove an item at a given index, and remove all data items. The sort button uses a **bubble sort** algorithm to sort the values the user inputs. Sorting is needed for probability as it allows for a better visualization of the quartiles.



```java
public void bubbleSort(ArrayList<Double> data) {
    boolean swapped = true;
    while (swapped) {
        swapped = false;
        for (int i = 1; i < data.size(); i++) {
            if (data.get(i - 1) > data.get(i)) {
                double temp = data.get(i);
                data.set(i, data.get(i - 1));
                data.set(i - 1, temp);
                swapped = true;
            }
        }
    }
}
```

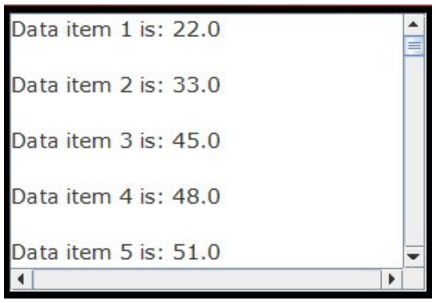
For example, below shows the data items before and after the sort button is clicked.

Before:



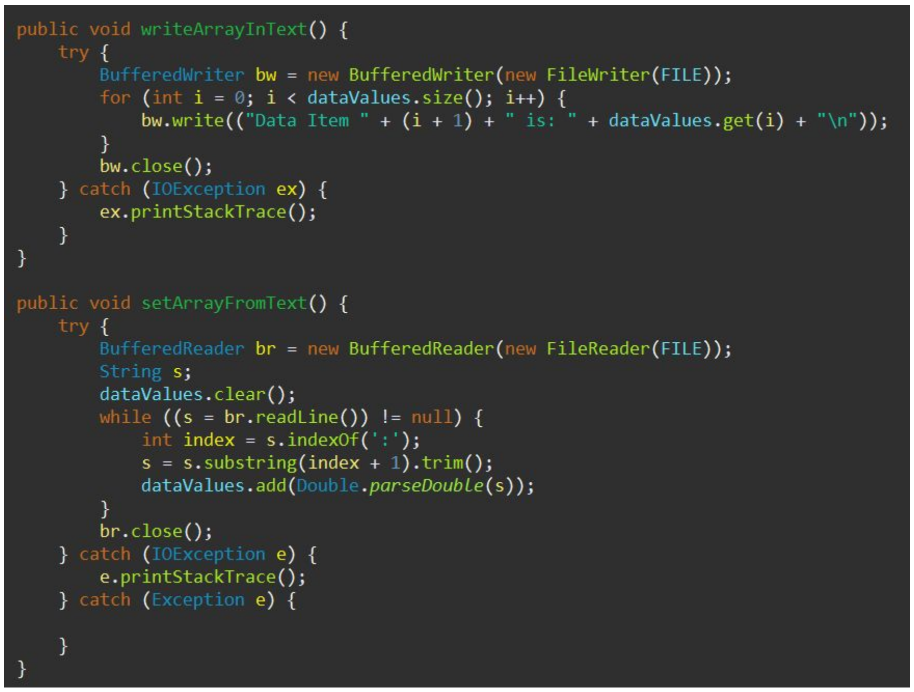
```
Data Item 1 is: 51.0

Data Item 2 is: 45.0

Data Item 3 is: 48.0

Data Item 4 is: 22.0

Data Item 5 is: 33.0
```

After:



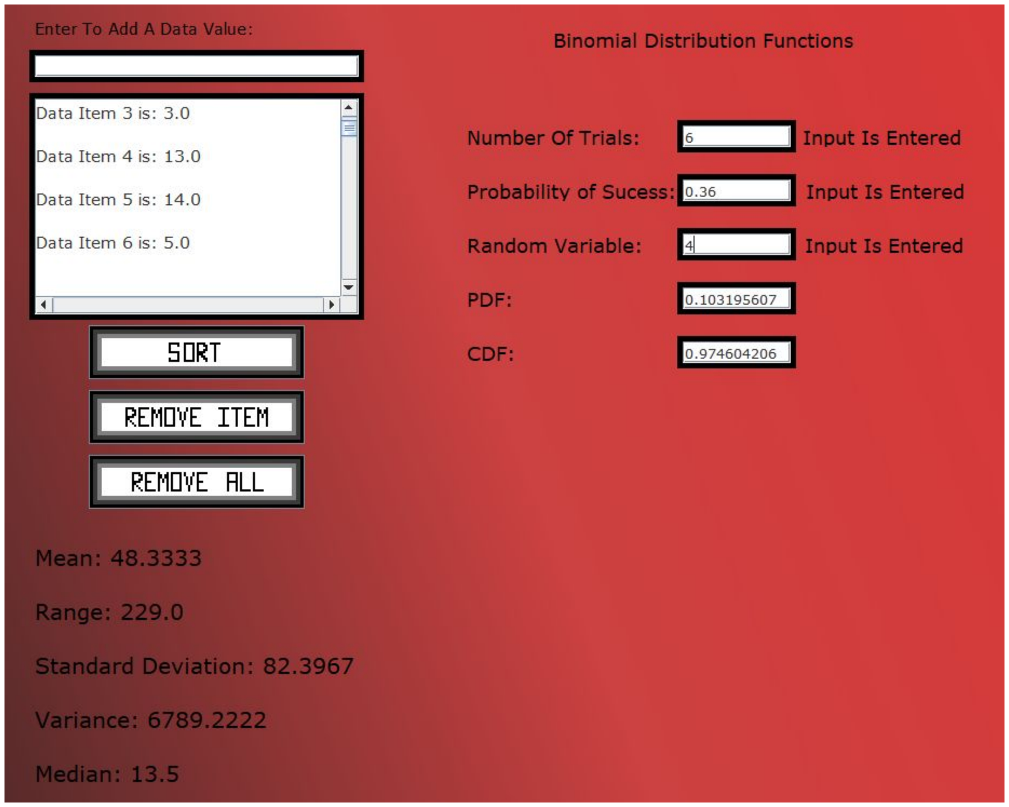
```
Data item 1 is: 22.0

Data item 2 is: 33.0

Data item 3 is: 45.0

Data item 4 is: 48.0

Data item 5 is: 51.0
```

As the values are added to the ArrayList, they are saved in a text file through the use of **BufferedWriter**. Thus, if the user inputs values then closes the program, the values are saved. The next time the program runs, the text file will be read using **BufferedReader** and displayed. The use of file input/output is very effective for the **StatisticsPanel** as users tend to have large lists of data to input when doing statistics problems for HL Math. Additionally, these statistics problems can have multiple parts so if a student closes the program, there is no need to re-add all the data, saving a lot of time. To the right is the code for the **BufferedReader** and **BufferedWriter**.



```java
public void writeArrayInText() {
    try {
        BufferedWriter bw = new BufferedWriter(new FileWriter(FILE));
        for (int i = 0; i < dataValues.size(); i++) {
            bw.write(("Data Item " + (i + 1) + " is: " + dataValues.get(i) + "\n"));
        }
        bw.close();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}

public void setArrayFromText() {
    try {
        BufferedReader br = new BufferedReader(new FileReader(FILE));
        String s;
        dataValues.clear();
        while ((s = br.readLine()) != null) {
            int index = s.indexOf(':');
            s = s.substring(index + 1).trim();
            dataValues.add(Double.parseDouble(s));
        }
        br.close();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (Exception e) {

    }
}
```

To evaluate the mean, range, and other properties of the data, math formulas were used. For example, the formula for the standard deviation for a given set of data was used to calculate the standard deviation. Recursion was used to evaluate factorials of numbers as that is very common in probability and used in the binomial distribution functions. These properties are updated each time a change to the data is made. This part was very simple to implement due to the use of math formulas.



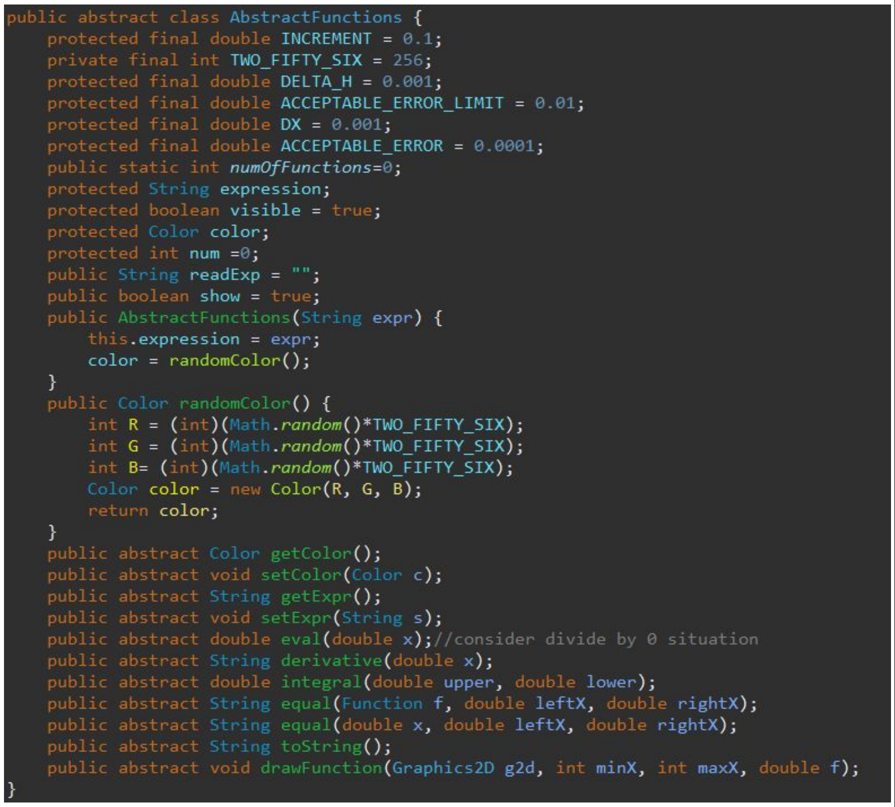
Data Item 3 is: 3.0
Data Item 4 is: 13.0
Data Item 5 is: 14.0
Data Item 6 is: 5.0

Number Of Trials: 6 Input Is Entered
Probability of Sucess: 0.36 Input Is Entered
Random Variable: 4 Input Is Entered
PDF: 0.103195607
CDF: 0.974604206

Mean: 48.3333
Range: 229.0
Standard Deviation: 82.3967
Variance: 6789.2222
Median: 13.5

AbstractFunctions

When creating the class for functions, it was necessary to be organized and plan out the methods that would be needed as subclasses will inherit functionalities from this class. Thus, an **abstract class** was created. Some of the notable abstract features that I wanted for the **abstract methods** were the **derivative()** method that takes the derivative at a given point, the **integral()** method that takes the integral given 2 bounds, the **equal()** method that returns the x-coordinate of the point of intersection of 2 functions, and the **drawFunction()** method. Additionally, the **static** integer **numOfFunctions** represents the number of functions that are present. It is incremented by 1 for each instantiation of a function and decreased by 1 for each removal of a function. The use of a static field is very effective as the variable **numOfFunctions** will still have a value of 0 since it is not specific to an object, but rather a class. The use of **encapsulation** is present as the variables are **protected** since the parent class needs to **inherit** the variables as well. Notice that below, the use of a random number generator is present to generate a color by randomly generating RGB values. This allows for ease of use as the user likely does not care about the colors of the functions, they just need to be distinct.

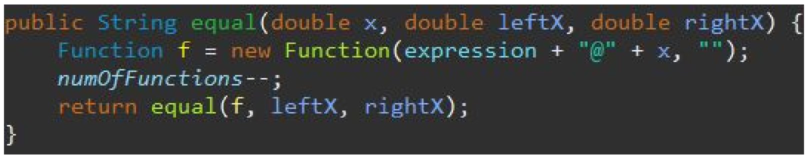


```java
public abstract class AbstractFunctions {
    protected final double INCREMENT = 0.1;
    private final int TWO_FIFTY_SIX = 256;
    protected final double DELTA_H = 0.001;
    protected final double ACCEPTABLE_ERROR_LIMIT = 0.01;
    protected final double DX = 0.001;
    protected final double ACCEPTABLE_ERROR = 0.0001;
    public static int numOfFunctions=0;
    protected String expression;
    protected boolean visible = true;
    protected Color color;
    protected int num =0;
    public String readExp = "";
    public boolean show = true;
    public AbstractFunctions(String expr) {
        this.expression = expr;
        color = randomColor();
    }
    public Color randomColor() {
        int R = (int)(Math.random()*TWO_FIFTY_SIX);
        int G = (int)(Math.random()*TWO_FIFTY_SIX);
        int B= (int)(Math.random()*TWO_FIFTY_SIX);
        Color color = new Color(R, G, B);
        return color;
    }
    public abstract Color getColor();
    public abstract void setColor(Color c);
    public abstract String getExpr();
    public abstract void setExpr(String s);
    public abstract double eval(double x);//consider divide by 0 situation
    public abstract String derivative(double x);
    public abstract double integral(double upper, double lower);
    public abstract String equal(Function f, double leftX, double rightX);
    public abstract String equal(double x, double leftX, double rightX);
    public abstract String toString();
    public abstract void drawFunction(Graphics2D g2d, int minX, int maxX, double f);
}
```

Functions

The **Functions** class **inherits** from the **AbstractFunctions** class and the abstract methods are developed. By inheriting from the parent class, code is reused for time efficiency.

Method overloading was used as there are two methods within the Functions class called equal(), one that takes another function as a parameter while one takes a double as a parameter.



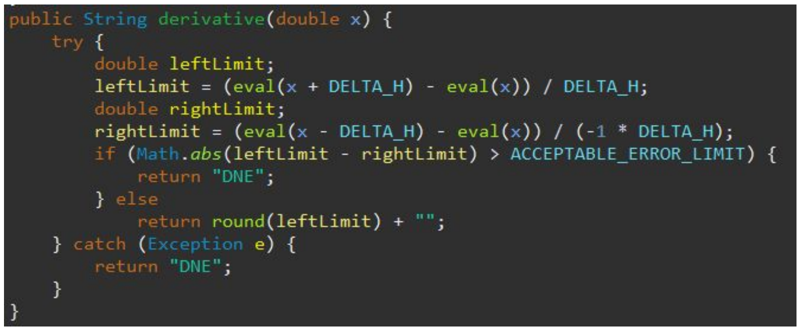
```java
public String equal(double x, double leftX, double rightX) {
    Function f = new Function(expression + "@" + x, "");
    numOfFunctions--;
    return equal(f, leftX, rightX);
}
```

derivative()

The derivative method returned the derivative of the function for a given x value. This was done by using the limit definition of a derivative at a single point.

$$f'(x) = \lim_{h \to 0} \frac{f(x+h)-f(x)}{h}$$

The value of h was very small, 0.001, which provides a reasonable approximation of the derivative. Also, this



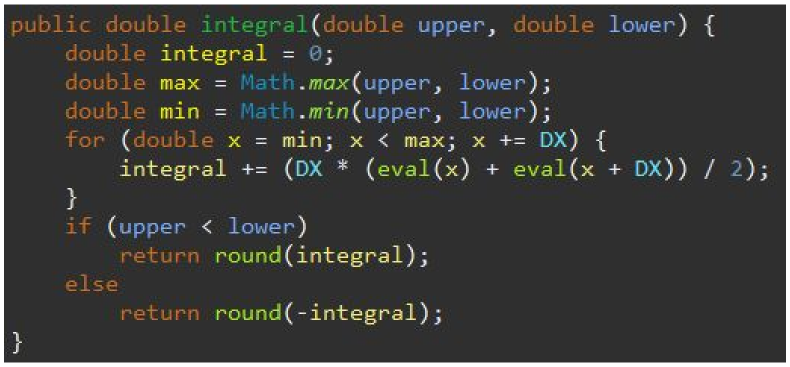
```java
public String derivative(double x) {
    try {
        double leftLimit;
        leftLimit = (eval(x + DELTA_H) - eval(x)) / DELTA_H;
        double rightLimit;
        rightLimit = (eval(x - DELTA_H) - eval(x)) / (-1 * DELTA_H);
        if (Math.abs(leftLimit - rightLimit) > ACCEPTABLE_ERROR_LIMIT) {
            return "DNE";
        } else
            return round(leftLimit) + "";
    } catch (Exception e) {
        return "DNE";
    }
}
```

limit was evaluated from the left and from the right, and if the two sides of the limit are not within a reasonable range, the return is “DNE”.

integral()
The integral method returns the integral of the function for a given upper and lower bounds. This was done by using the limit definition of an integral at a single point.

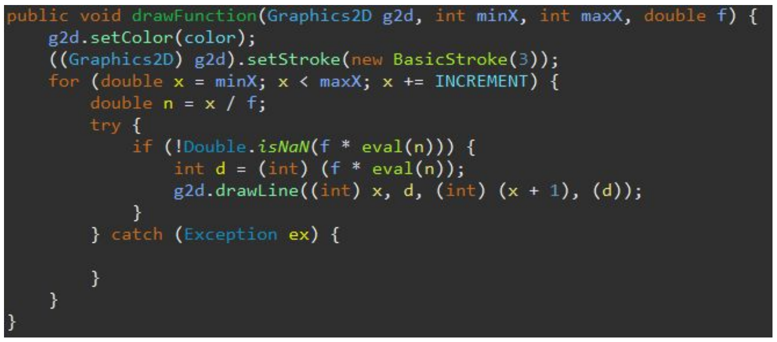
$$F(x) = \lim_{dx \to 0} \sum_{x=lower}^{upper} \frac{f(x)+f(x+dx)}{2} dx$$

The value of dx was very small, 0.001, which provides a reasonable approximation of the derivative.



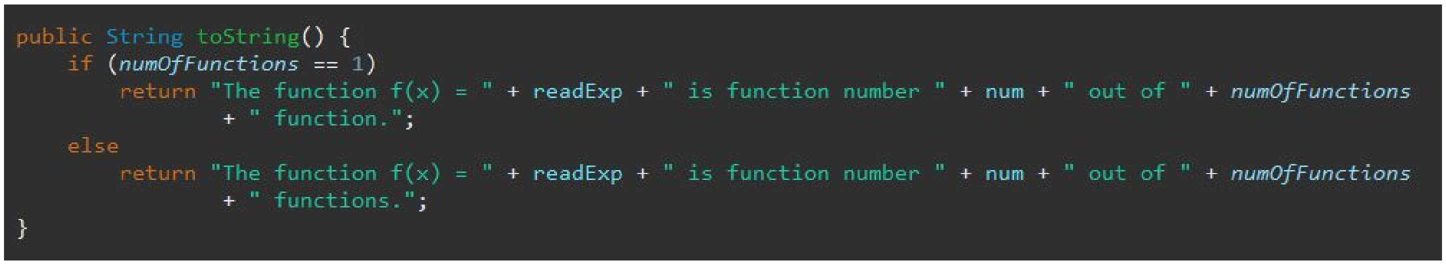
```java
public double integral(double upper, double lower) {
    double integral = 0;
    double max = Math.max(upper, lower);
    double min = Math.min(upper, lower);
    for (double x = min; x < max; x += DX) {
        integral += (DX * (eval(x) + eval(x + DX)) / 2);
    }
    if (upper < lower)
        return round(integral);
    else
        return round(-integral);
}
```

drawFunction()
This method draws the function and is done by drawing many, short line segments, giving the effect of a function.



```java
public void drawFunction(Graphics2D g2d, int minX, int maxX, double f) {
    g2d.setColor(color);
    ((Graphics2D) g2d).setStroke(new BasicStroke(3));
    for (double x = minX; x < maxX; x += INCREMENT) {
        double n = x / f;
        try {
            if (!Double.isNaN(f * eval(n))) {
                int d = (int) (f * eval(n));
                g2d.drawLine((int) x, d, (int) (x + 1), (d));
            }
        } catch (Exception ex) {

        }
    }
}
```

toString()
This method overrides the predefined **toString()** method in the Object class. The default method prints out the memory address which is not very useful for the user. Thus, this method is **overridden** and will return the expression for the function and its function number out of the total number of functions. This also highlighted the need for the static variable **numOfFunctions**.



```java
public String toString() {
    if (numOfFunctions == 1)
        return "The function f(x) = " + readExp + " is function number " + num + " out of " + numOfFunctions
                + " function.";
    else
        return "The function f(x) = " + readExp + " is function number " + num + " out of " + numOfFunctions
                + " functions.";
}
```

eval()
The **eval()** method evaluates the function at a given point. It uses the same algorithm as the ScientificPanel **eval()** method.

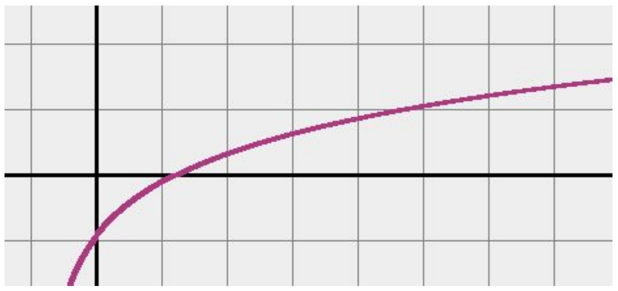
equal()

The **equal()** method outputs the x-coordinate of the point of intersection of the function and a parameter function, given the left bound and the right bound. Say the parameter function is $h(x)$ and the instantiated function is $f(x)$, a new function $g(x) = h(x)-f(x)$ can be formed. The x-coordinate such that $g(x)=0$ is the same x-coordinate such that $h(x)=f(x)$. This x coordinate will be called a. The algorithm used to solve for a will be **binary search** and its advantages will be mentioned later on. To continue with this algorithm, $g(x)$ is assumed to be continuous and $h(x)$ cannot be tangent to $f(x)$ at $x=a$.

Case 1: There is exactly one x value such that $g(x) = 0$.

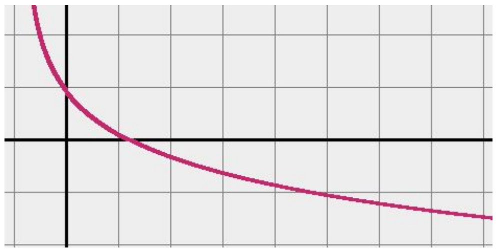
Case 1.1: $g(x)<0$ for $x<a$ and $g(x)>0$ for $x>a$.

For example, the function may look like the one on the right. The x coordinate of the left bound will be called **minX** and the x coordinate of the right bound will be called **maxX**. Since there is exactly one x value such that $h(x) = 0$ and $g(x)<0$ for $x<a$ and $g(x)>0$ for $x>a$, g(**maxX**)>0 and g(**minX**)<0. Since g(x) is a continuous function, the function must pass through the x-axis, which will be at x=a. The middle value of **minX** and **maxX** will be taken and will be called **midX**. If g(**midX**)>0, that means a<**midX**. In this case, **maxX** will equal **midX** and **midX** will be recalculated. If g(**midX**)<0, that means a>**midX**. In this case, **minxX** will equal **midX** and **midX** will be recalculated. Each time this is done, the range of **maxX** and **minX** will decrease and **midX** will get closer and closer to a. However, it is possible that g(**midX**) never exactly equals 0 due to the infinite number of real numbers between any two distinct numbers. Thus, when the difference between g(**midX**) and 0 is acceptably small, the algorithm will stop and return the value of **midX** as the intersection of $g(x)=0$. In other words, h(**midX**)≅f(**midX**) and a rounded **midX** to 5 decimal places will be returned as the x coordinate of the point of intersection.



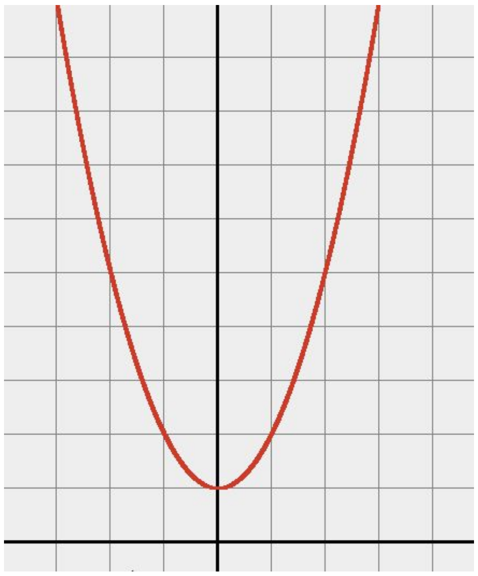
Case 1.2: $g(x)>0$ for $x<a$ and $g(x)<0$ for $x>a$.

For example, the function may look like the one on the right. The same argument can be made except a small adjustment has to be made. If g(**midX**)>0, that means a>**midX** as opposed to the previous case where if g(**midX**)>0, then a<**midX**. Similarly, if g(**midX**)<0, that means a<**midX**.



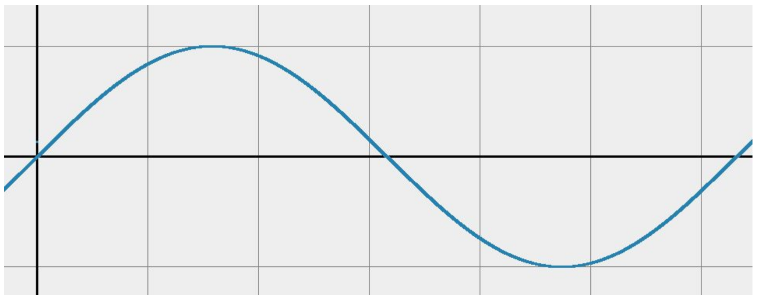
Case 2: There are no x values such that h(x) = 0.

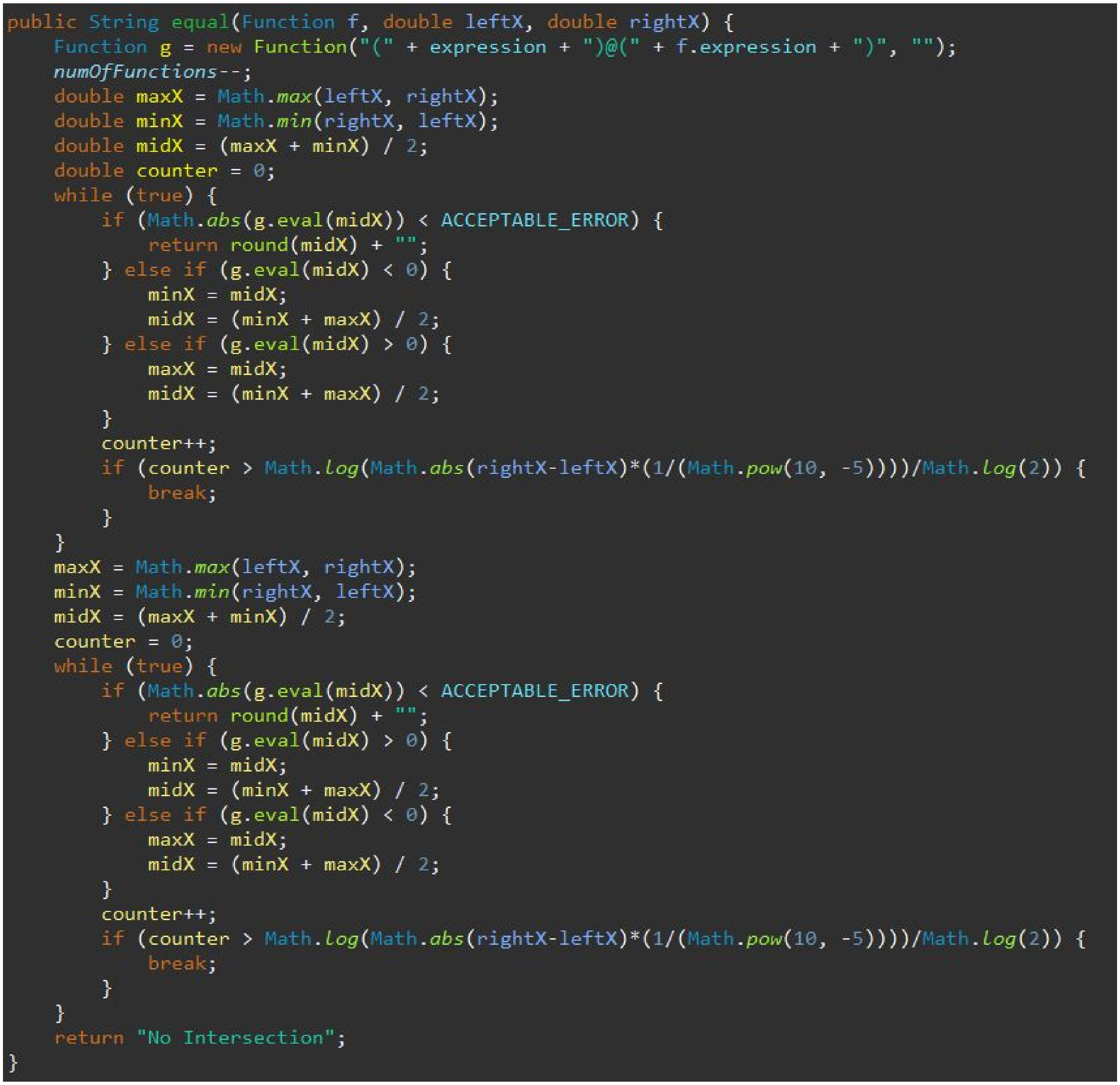
For example, the function may look like the one on the right. In this case, the previously mentioned algorithm would keep running forever and never terminate as g(**midX**) never approaches 0. Thus, the algorithm will terminate after a certain number of iterations. Recall that the algorithm in Case 1 terminates if the difference between g(**midX**) and 0 is acceptably small. In other words, the program terminates when Math.abs(g.eval(**midX**))< **ACCEPTABLE_ERROR**. Since the returned x-coordinate is rounded to 5 decimal places. The smallest increment in an x value that is possible is 10^(-5). Thus, there are a total of Math.abs(**rightX-leftX**)/Math.pow(10, -5) possible x-values, where **leftX** is the parameter of the method that provides the x-coordinate of the left bound and **rightX** is the parameter of the method that provides the x-coordinate of the right bound. Recall that binary search takes log base 2 of **n** time where **n** is the number of items. In this case, **n=**Math.abs(**rightX-leftX**)/Math.pow(10,-5). Thus the program will terminate after log base 2 of Math.abs(**rightX-leftX**)/Math.pow(10, -5) iterations and will return “No Intersection”.



Case 3: There are more than one x value such that g(x) = 0.

For example, the function may look like the one on the right. The exact same argument can be made as in Case 1. There is no modification to be made as the program will just output one of the many x values such that g(x)=0. This is because the binary search algorithm will decrease the size of the interval of search through each iteration, and eventually, there will only be one x value that satisfies g(x)=0 in the interval.

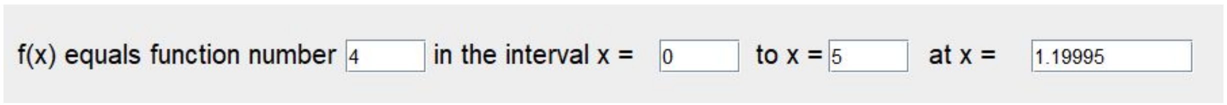




```java
public String equal(Function f, double leftX, double rightX) {
    Function g = new Function("(" + expression + ")@(" + f.expression + ")", "");
    numOfFunctions--;
    double maxX = Math.max(leftX, rightX);
    double minX = Math.min(rightX, leftX);
    double midX = (maxX + minX) / 2;
    double counter = 0;
    while (true) {
        if (Math.abs(g.eval(midX)) < ACCEPTABLE_ERROR) {
            return round(midX) + "";
        } else if (g.eval(midX) < 0) {
            minX = midX;
            midX = (minX + maxX) / 2;
        } else if (g.eval(midX) > 0) {
            maxX = midX;
            midX = (minX + maxX) / 2;
        }
        counter++;
        if (counter > Math.log(Math.abs(rightX-leftX)*(1/(Math.pow(10, -5))))/Math.log(2)) {
            break;
        }
    }
    maxX = Math.max(leftX, rightX);
    minX = Math.min(rightX, leftX);
    midX = (maxX + minX) / 2;
    counter = 0;
    while (true) {
        if (Math.abs(g.eval(midX)) < ACCEPTABLE_ERROR) {
            return round(midX) + "";
        } else if (g.eval(midX) > 0) {
            minX = midX;
            midX = (minX + maxX) / 2;
        } else if (g.eval(midX) < 0) {
            maxX = midX;
            midX = (minX + maxX) / 2;
        }
        counter++;
        if (counter > Math.log(Math.abs(rightX-leftX)*(1/(Math.pow(10, -5))))/Math.log(2)) {
            break;
        }
    }
    return "No Intersection";
}
```

For example,

$f(x) = ln(\frac{x}{2} + 0.4)$ and $function\ number\ 4 =- ln(\frac{x}{2} + 0.4)$



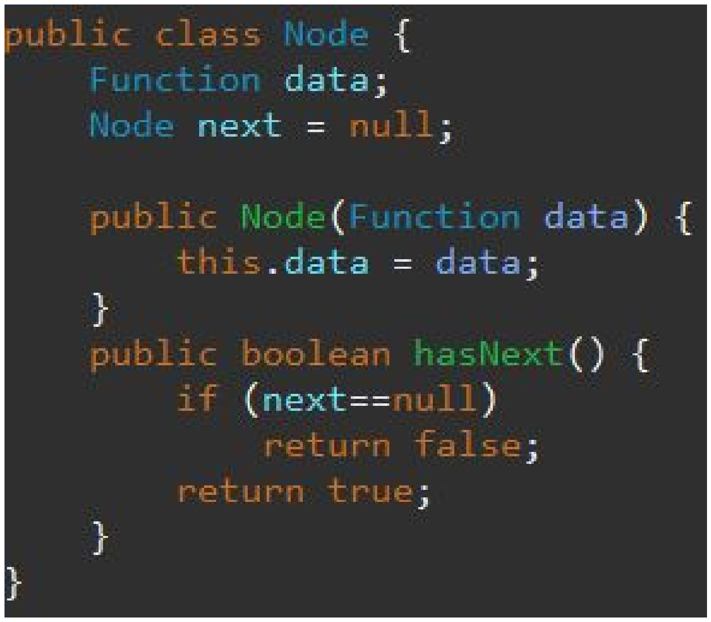
f(x) equals function number 4 in the interval x = 0 to x = 5 at x = 1.19995

The use of binary search is that it allows the program to guess to answer to a very high degree of accuracy. Additionally, by halving the interval of search each time, this algorithm is very time efficient and takes O(log(n)) time. Binary search requires the values to be sorted, but since the algorithm assumes the functions are continuous, the values will already be sorted. The program simply does not know whether the function is increasing or decreasing so there are two binary searches, but the time complexity still remains as O(log(n)) time.

SinglyLinkedList

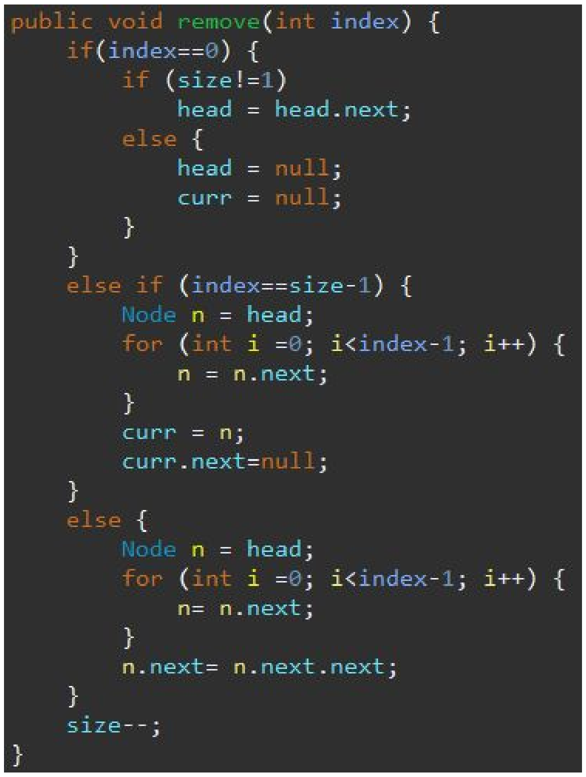
A singly linked list was created to store the functions. As the number of functions is constantly changing depending on if the user adds or removes functions, the variable size of a linked list is very effective to store the data. Also, insertions and deletions of nodes within the linked list can easily be performed.

First, a **Node** class has to be made which represents each node in the linked list. Each node contains the data for one function and the node that it points to.

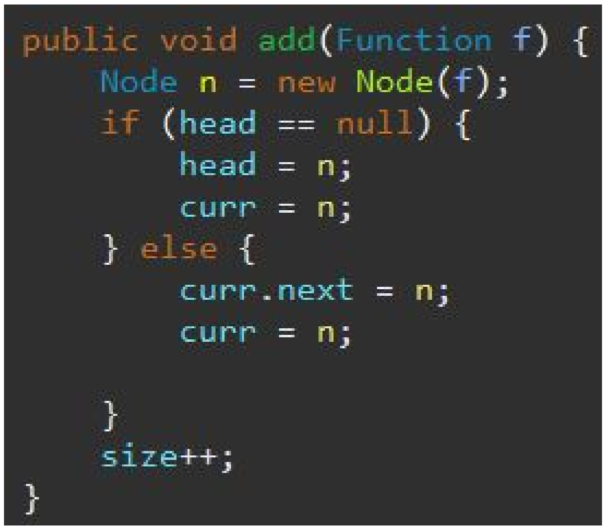
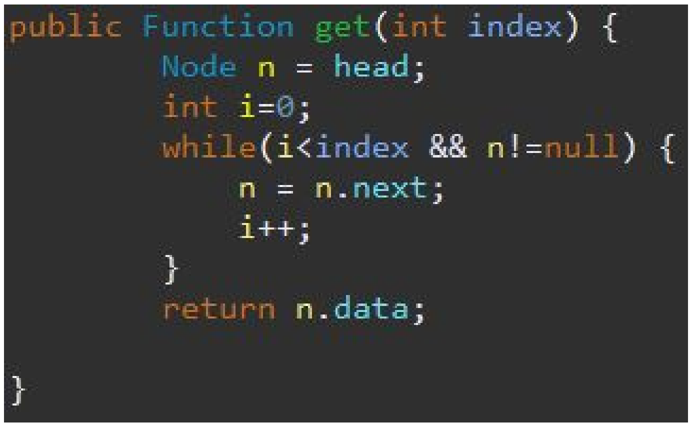


```java
public class Node {
    Function data;
    Node next = null;

    public Node(Function data) {
        this.data = data;
    }
    public boolean hasNext() {
        if (next==null)
            return false;
        return true;
    }
}
```

The linked list has nodes head and tail along with a size. It has basic methods such as adding an element, getting the data at a given index in the linked list, removing an element, and getting the size. To the left is the **remove()** method which removes the element at a given index.



```java
public void remove(int index) {
    if(index==0) {
        if (size!=1)
            head = head.next;
        else {
            head = null;
            curr = null;
        }
    }
    else if (index==size-1) {
        Node n = head;
        for (int i =0; i<index-1; i++) {
            n = n.next;
        }
        curr = n;
        curr.next=null;
    }
    else {
        Node n = head;
        for (int i =0; i<index-1; i++) {
            n= n.next;
        }
        n.next= n.next.next;
    }
    size--;
}
```

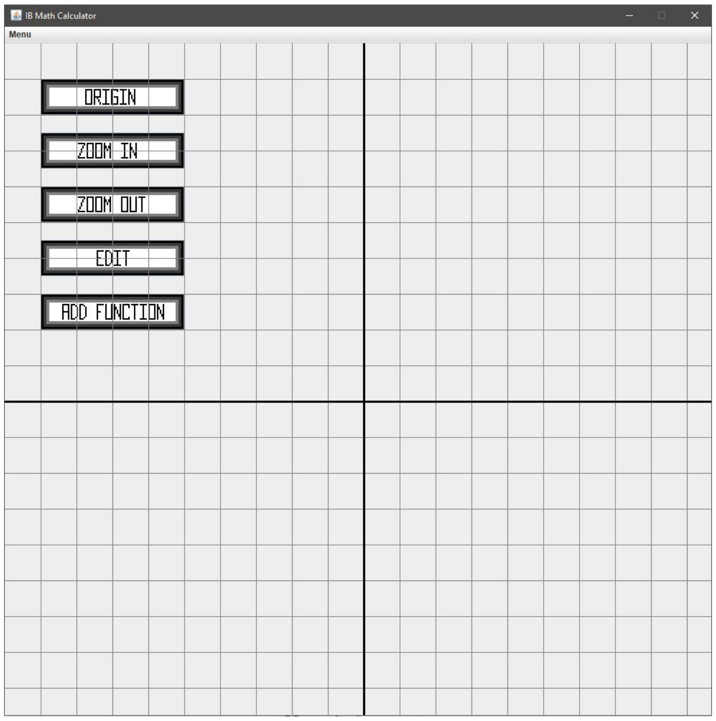
To the left is the **add()** method which adds a Function object to the end of the linked list.	
To the left is the **get()** method which returns the Function object at a given index.	

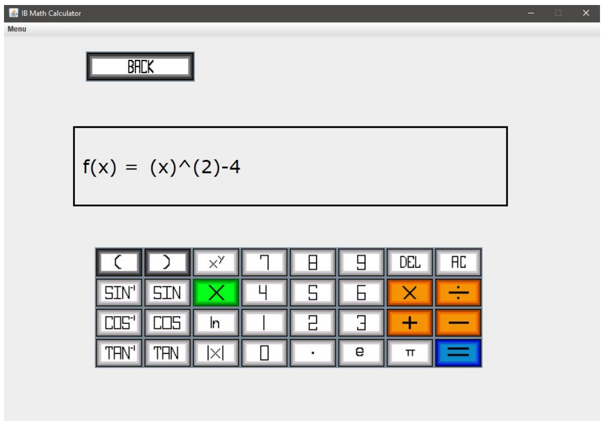
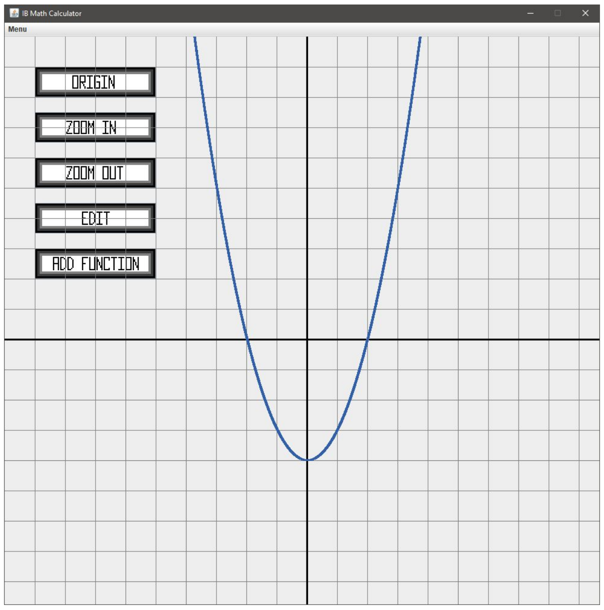
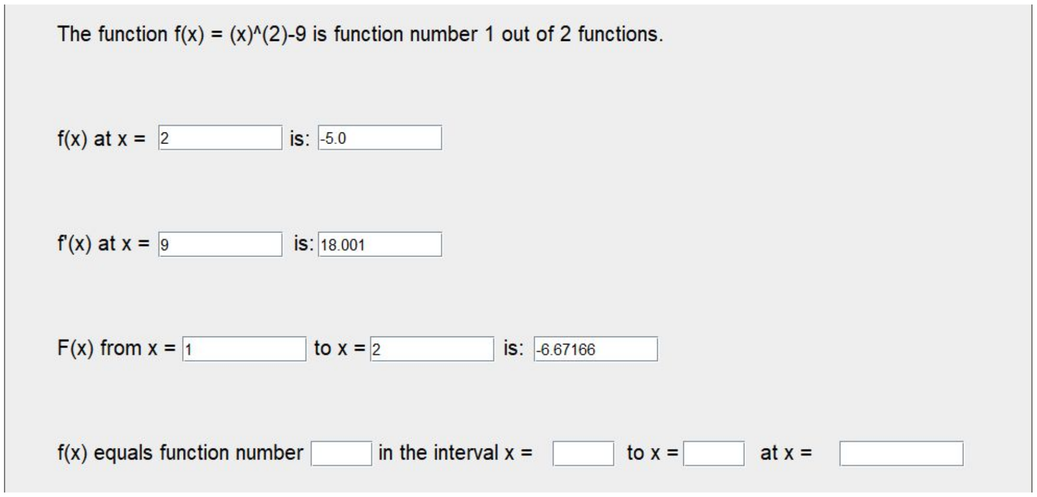
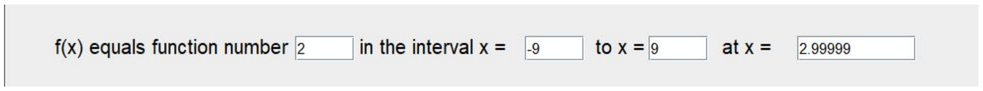
```
public void add(Function f) {
    Node n = new Node(f);
    if (head == null) {
        head = n;
        curr = n;
    } else {
        curr.next = n;
        curr = n;

    }
    size++;
}
```

```
public Function get(int index) {
        Node n = head;
        int i=0;
        while(i<index && n!=null) {
            n = n.next;
            i++;
        }
        return n.data;

}
```

Graphing Panel

This JPanel graphs the functions that the user enters.

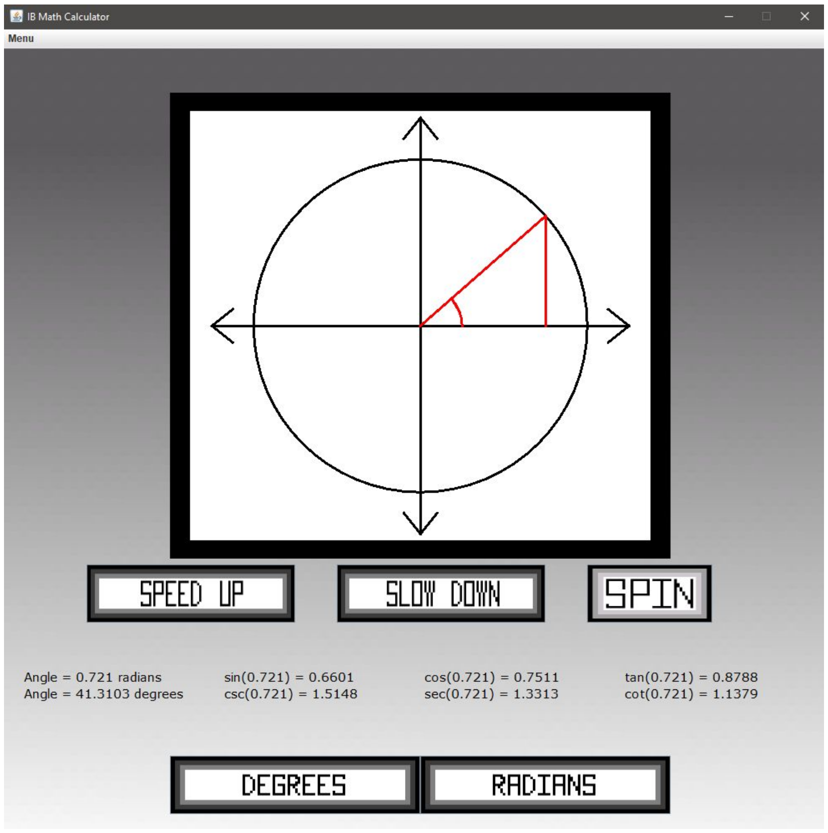
JButtons and KeyListener were used. There is a “zoom in” button, “zoom out” button, return to “origin” button, and an “edit function” button. These buttons are see-through to ensure they do not block much of the graph. Arrow keys can also be used to move the screen up, down, left, or right.	

In this panel, there is a linked list of that store **Function** objects. To add functions, the “add function” button can be pressed to take the user to the **AddFunctionPanel** where the user can add a math expression string for a function in terms of x. The algorithm used to input the function is the same as the scientific panel. The main difference is the addition of the “x” term whose button is green.	 IB Math Calculator Menu BACK ( ) x^y 7 8 9 DEL AC SIN⁻¹ SIN X 4 5 6 × ÷ COS⁻¹ COS ln 1 2 3 + — TAN⁻¹ TAN |x| 0 . e π = f(x) = (x)^(2)-4
When the user clicks the blue equals button, the function is added to the linked list and the user is taken back to the graphing panel screen with the function drawn.	 IB Math Calculator Menu ORIGIN ZOOM IN ZOOM OUT EDIT ADD FUNCTION
If the user clicks the button “edit functions”, they are taken to the **EditFunctionsPanel** which contains many JButtons, JLabels, and JTextFields as shown to the right. Here, the user can enter numbers into the text fields and the program will evaluate them accordingly.	 The function f(x) = (x)^(2)-9 is function number 1 out of 2 functions. f(x) at x = 2 is: -5.0 f'(x) at x = 9 is: 18.001 F(x) from x = 1 to x = 2 is: -6.67166 f(x) equals function number in the interval x = to x = at x =
For the image above, if a second function, g(x) = 5 is added and the user can find the point of intersection between f(x) and g(x) using the **equals()** in the **Function** class.	 f(x) equals function number 2 in the interval x = -9 to x = 9 at x = 2.99999

TrigPanel

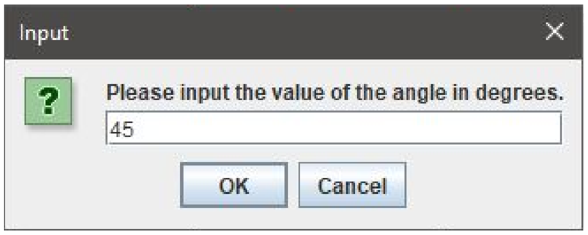
The **TrigPanel** is a JPanel that allows the user to visualize the geometry of trigonometric functions.

For the **TrigPanel**, there is a unit circle with a triangle within it and the angle in the standard position is labelled. The unit circle and the triangle is drawn using many lines and some basic trigonometric math.



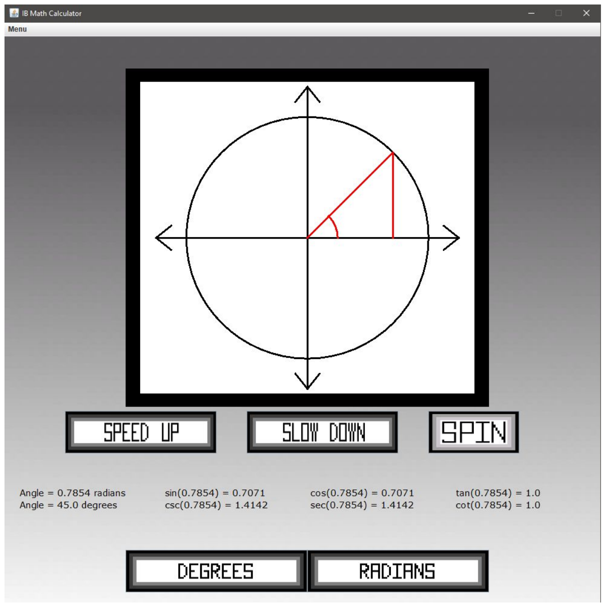
```
Angle = 0.721 radians        sin(0.721) = 0.6601     cos(0.721) = 0.7511     tan(0.721) = 0.8788
Angle = 41.3103 degrees      csc(0.721) = 1.5148     sec(0.721) = 1.3313     cot(0.721) = 1.1379
```

There are JButtons such as that can speed up the spin, slow down the spin, or stop/start the spin. Additionally, to the right are the values of trigonometric functions at the current angle. To the right are two buttons where the user can choose to enter an angle in degrees or radians and the angle will be drawn.



Please input the value of the angle in degrees.

45



```
Angle = 0.7854 radians      sin(0.7854) = 0.7071     cos(0.7854) = 0.7071     tan(0.7854) = 1.0
Angle = 45.0 degrees        csc(0.7854) = 1.4142     sec(0.7854) = 1.4142     cot(0.7854) = 1.0
```

Techniques Used	
Technique	Where Technique Is Used
Stacks	StringStack and DoubleStack
Recursion	eval() and factorial()
LinkedList	Self-implemented SinglyLinkedList
Abstract Class	AbstractFunctions Class
Inheritance	Functions inherit from AbstractFunctions
Encapsulation	Use of public, private, and protected
Static Variables	numOfFunctions variable in AbstractFunctions
Method Overriding	Functions toString() methods outputs the expression for the function
Method Overloading	There are two equals() methods in the Function class, but with different parameters
Binary Search	Used in finding the point of intersection for 2 functions
File I/O	Saves the data values that are entered
Sorting	Bubble sort to sort data values
String Handling	Used in eval() method
Random Number Generator	Colors of Functions were randomly generated
Error Handling	User input was put in try-catch
Reasons as to why these techniques are adequate for the task were previously mentioned.	

Imports

```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import javax.swing.*;
import java.io.*;
import java.util.*;
```

See appendix B for UML Diagram.
See appendix D for references embedded in source code.

Sources Used

Oracle. (2020). *Class Graphics2D*. Retrieved on December 29, 2020 from https://docs.oracle.com/javase/7/docs/api/java/awt/Graphics2D.html

Tutorialspoint. (2020). *Java - Files and I/O*. Retrieved on December 21, 2020 from https://www.tutorialspoint.com/java/java_files_io.htm

Word Count: 1254