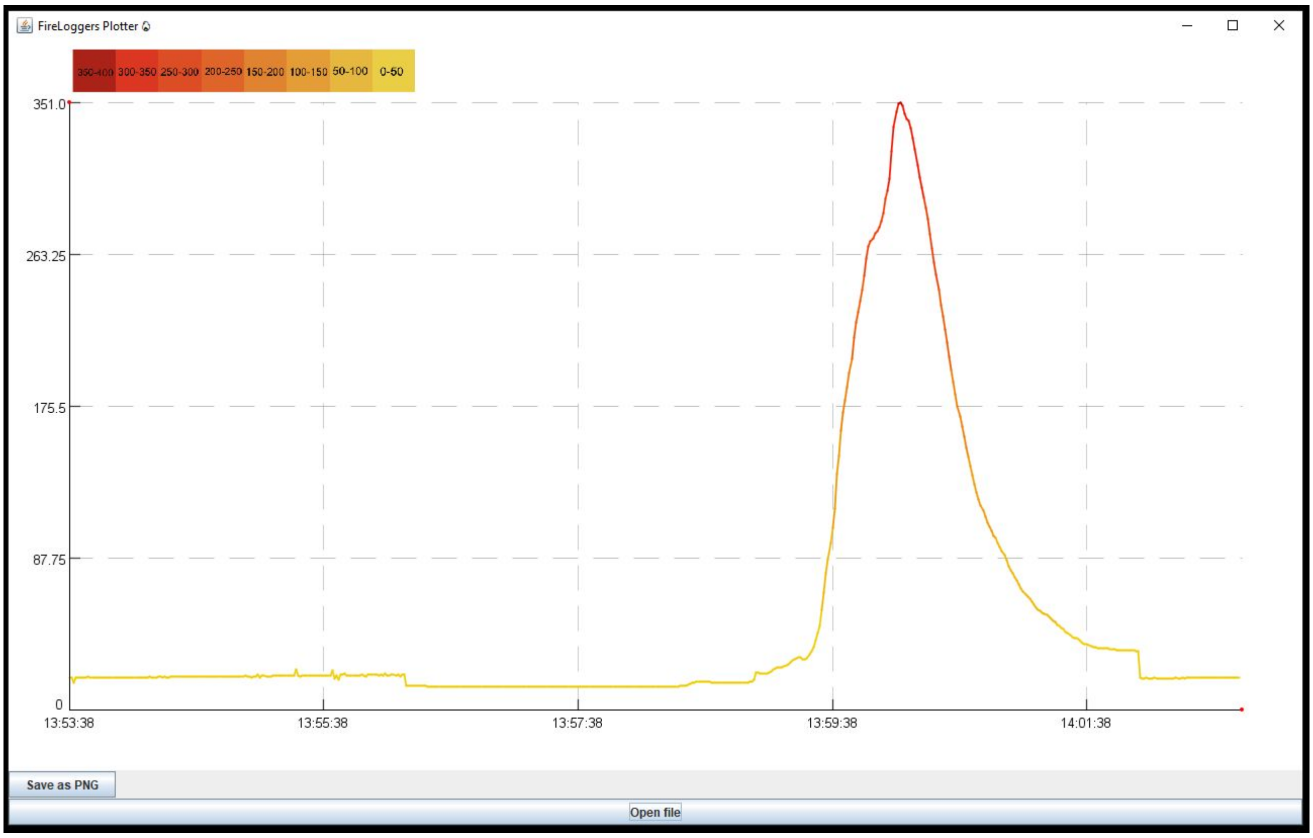
FireLoggers Plotter 🔥

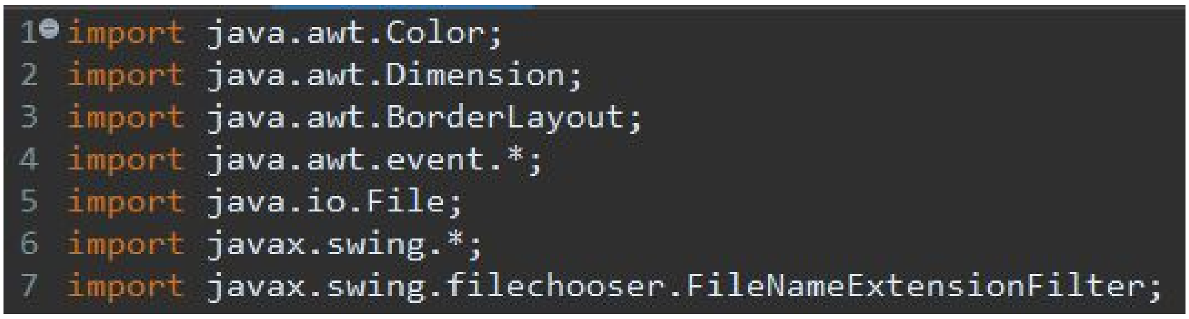
By María Gonzalez, Daniel Piñera, Daniel López & Maixent Andres.



🌐 https://github.com/Bimo99B9/FireLoggersPlotter

<u>SwingUI -></u>

We have two programs, one for the aesthetics of the window that will be on the screen and the other one the aesthetics of the graph and the information that will appear on it.



```java
import java.awt.Color;
import java.awt.Dimension;
import java.awt.BorderLayout;
import java.awt.event.*;
import java.io.File;
import javax.swing.*;
import javax.swing.filechooser.FileNameExtensionFilter;
```

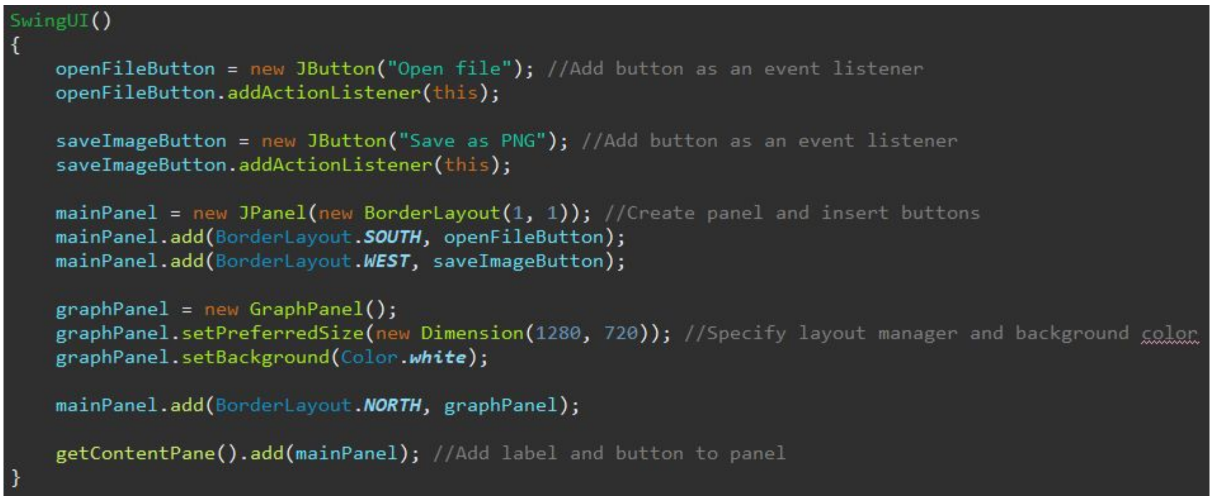
In the first one we will choose the dimension of the window that the user will have in his screen and also the fact that he will have different buttons. First we will add all the java imports that we will need to create a beautiful interface:

Here we will add seven imports to our java program (respectively) :

- For adding colors
- For controlling dimension of the screen
- For controlling the layout of the border of the screen
- For adding the vocabulary event that will help us to write easily
- For helping to use the files that we will choose
- For
- For helping to choose the file that we will analyze

After that we can start to write what we will have on our screen.

When we start the program we need an “Open file” button and a “Save as PNG” button, to open and to save the graph and all the information as a PNG file.



```java
SwingUI()
{
    openFileButton = new JButton("Open file"); //Add button as an event listener
    openFileButton.addActionListener(this);

    saveImageButton = new JButton("Save as PNG"); //Add button as an event listener
    saveImageButton.addActionListener(this);

    mainPanel = new JPanel(new BorderLayout(1, 1)); //Create panel and insert buttons
    mainPanel.add(BorderLayout.SOUTH, openFileButton);
    mainPanel.add(BorderLayout.WEST, saveImageButton);

    graphPanel = new GraphPanel();
    graphPanel.setPreferredSize(new Dimension(1280, 720)); //Specify layout manager and background color
    graphPanel.setBackground(Color.white);

    mainPanel.add(BorderLayout.NORTH, graphPanel);

    getContentPane().add(mainPanel); //Add label and button to panel
}
```

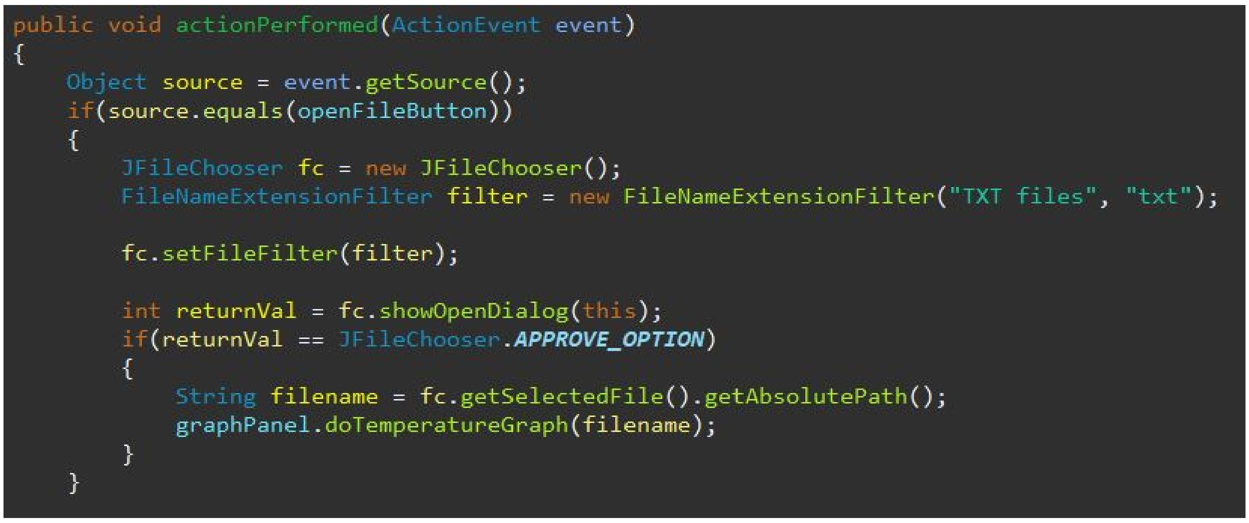
We also wrote a few lines that will make the open file appear in the “SOUTH”, so at the bottom of the border, and the save as PNG in the “WEST”, so at the left side of the border that we create. They will appear in the panel that we created before.

We also have to delimit the dimension of the screen and its colour. We chose a resolution of 1280x720 and a white color for the background. Then we add the button and the label to the screen with the last line in the last image.

The “actionPerformed” method indicates to the program what to do when one of the two buttons is pressed, so we have an “if” related to each button.

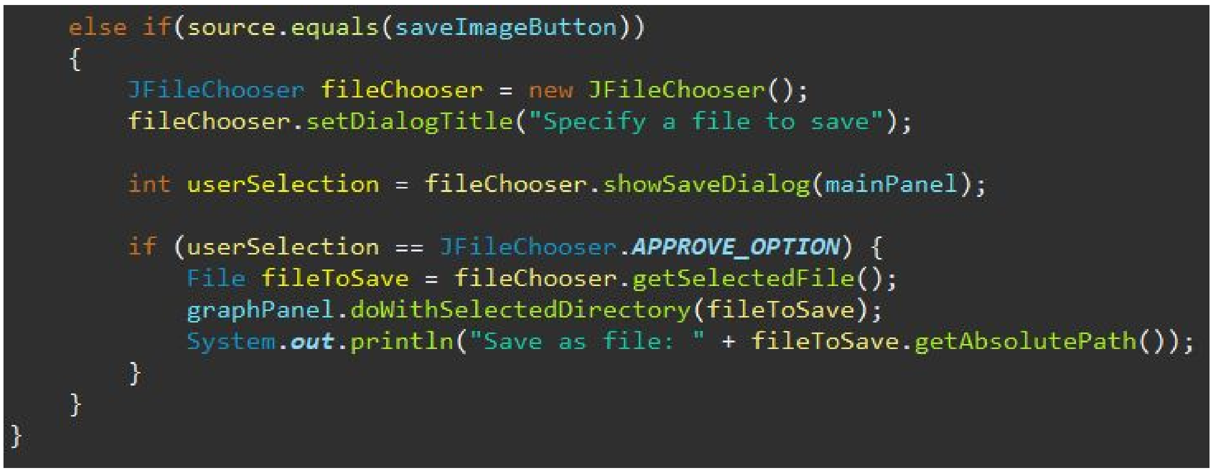
When the first button is pressed, it opens a window to search in all the files that are TXT files, ignoring the other types of files.

We still have the button in the program after choosing one document, to generate another graph with a different file.



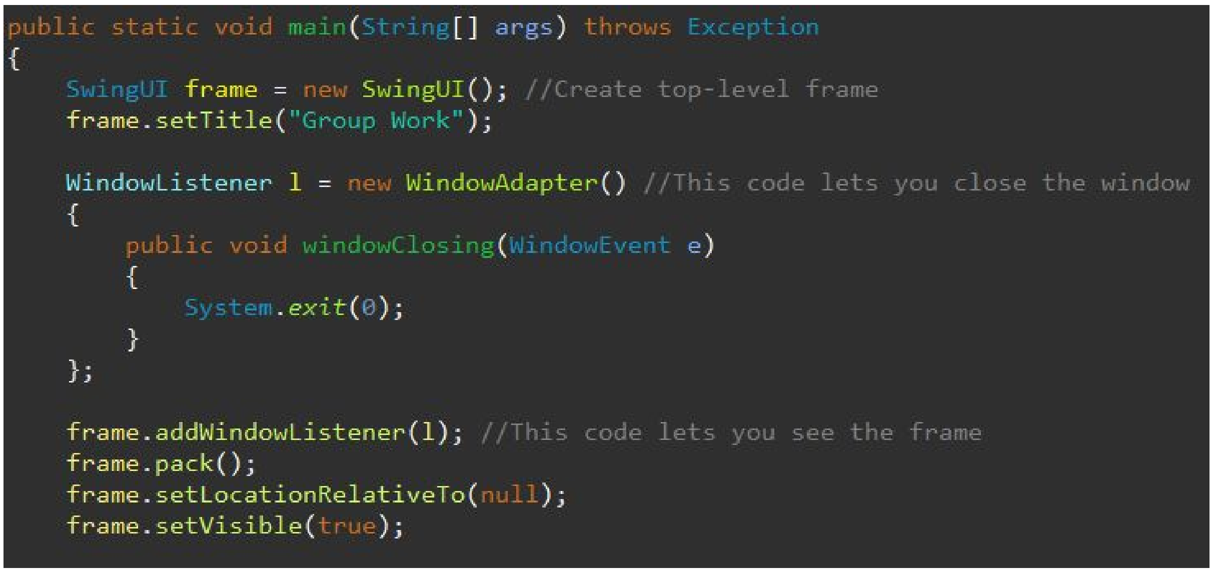
```java
public void actionPerformed(ActionEvent event)
{
    Object source = event.getSource();
    if(source.equals(openFileButton))
    {
        JFileChooser fc = new JFileChooser();
        FileNameExtensionFilter filter = new FileNameExtensionFilter("TXT files", "txt");

        fc.setFileFilter(filter);

        int returnVal = fc.showOpenDialog(this);
        if(returnVal == JFileChooser.APPROVE_OPTION)
        {
            String filename = fc.getSelectedFile().getAbsolutePath();
            graphPanel.doTemperatureGraph(filename);
        }
    }
```

The second one for saving the screen as a PNG will open our documents files and we will choose the path and give it a name. The window that will be opened by pressing this button will be named “Specify a file to save”. This part of the program only gets the path to save the image.



```java
    else if(source.equals(saveImageButton))
    {
        JFileChooser fileChooser = new JFileChooser();
        fileChooser.setDialogTitle("Specify a file to save");

        int userSelection = fileChooser.showSaveDialog(mainPanel);

        if (userSelection == JFileChooser.APPROVE_OPTION) {
            File fileToSave = fileChooser.getSelectedFile();
            graphPanel.doWithSelectedDirectory(fileToSave);
            System.out.println("Save as file: " + fileToSave.getAbsolutePath());
        }
    }
}
```

In the SwingUI class we have the main program code. Here we create the panel, the listener of the window and start the program



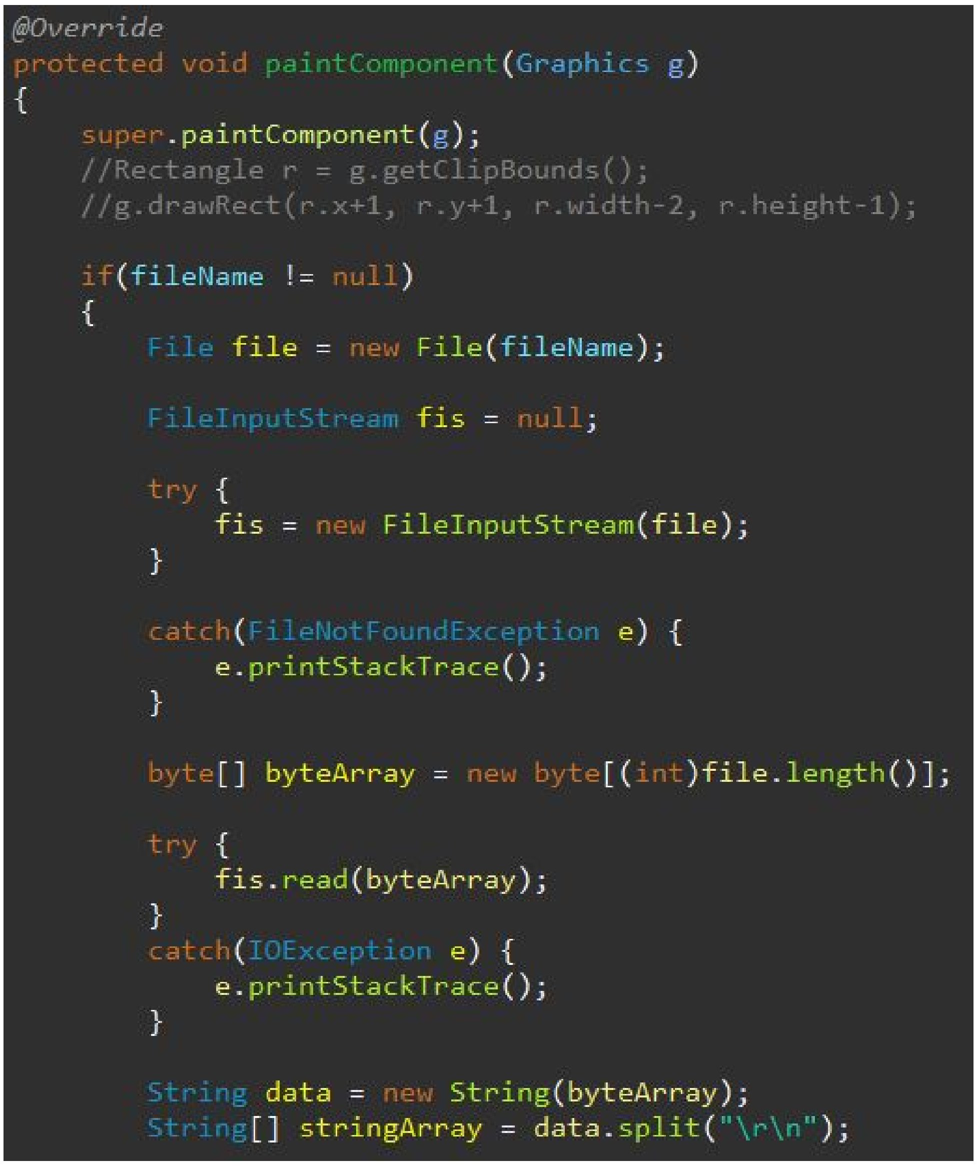
```java
public static void main(String[] args) throws Exception
{
    SwingUI frame = new SwingUI(); //Create top-level frame
    frame.setTitle("Group Work");

    WindowListener l = new WindowAdapter() //This code lets you close the window
    {
        public void windowClosing(WindowEvent e)
        {
            System.exit(0);
        }
    };

    frame.addWindowListener(l); //This code lets you see the frame
    frame.pack();
    frame.setLocationRelativeTo(null);
    frame.setVisible(true);
```

<u>**GraphPanel ->**</u>

In this class we have the code that generates the graph when a not null filename is given. The main method is paintComponent, but outside we have different methods that we needed while writing the paintComponent.

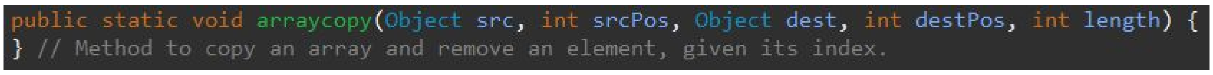
The variable “file” is of the class File, and then we create an object of the type FileInputStream in order to get the information from the txt file and we create a new empty array called byteArray that has the same size as the file(counting the number of lines). With the fis.read(byteArray), it reads up to byteArray.length bytes of data from this input stream into an array of bytes.

Then we generate the String “data” with all the info in the txt file, and then we split it into a new String[] “stringArray” where each position is a line from the file.

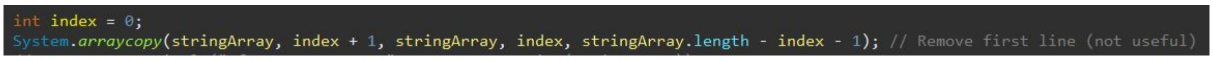


```
@Override
protected void paintComponent(Graphics g)
{
    super.paintComponent(g);
    //Rectangle r = g.getClipBounds();
    //g.drawRect(r.x+1, r.y+1, r.width-2, r.height-1);

    if(fileName != null)
    {
        File file = new File(fileName);

        FileInputStream fis = null;

        try {
            fis = new FileInputStream(file);
        }

        catch(FileNotFoundException e) {
            e.printStackTrace();
        }

        byte[] byteArray = new byte[(int)file.length()];

        try {
            fis.read(byteArray);
        }
        catch(IOException e) {
            e.printStackTrace();
        }

        String data = new String(byteArray);
        String[] stringArray = data.split("\r\n");
```

Then we use the created method “arraycopy” to be able to remove the first line of the file given its index, which is not important for plotting the graph.

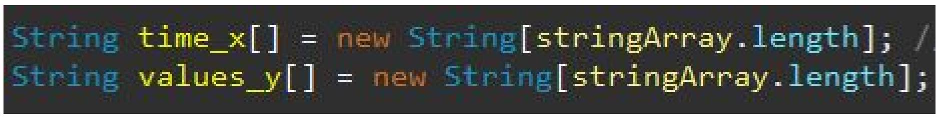


```java
public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length) {
} // Method to copy an array and remove an element, given its index.
```



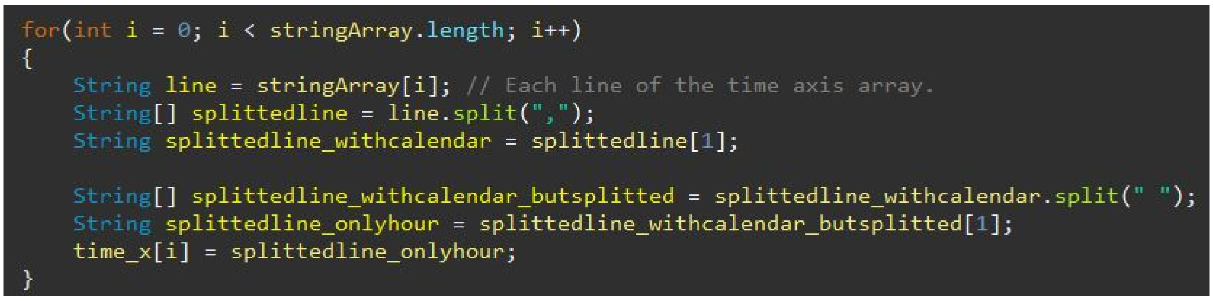
```java
int index = 0;
System.arraycopy(stringArray, index + 1, stringArray, index, stringArray.length - index - 1); // Remove first line (not useful)
```

Now we create two empty String arrays(time_x and values_y) with the same length as the stringArray, one of them is to store the time and the other one is used to store the temperatures.



```java
String time_x[] = new String[stringArray.length]; /
String values_y[] = new String[stringArray.length];
```

For the first string(time_x), we needed to split each line with the commas, and take the position of index=1 of the string array, which corresponds to the date and time of the measurement.

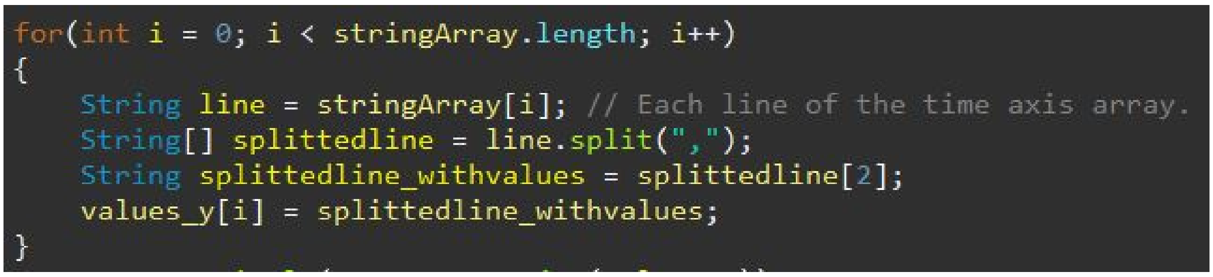


```java
for(int i = 0; i < stringArray.length; i++)
{
    String line = stringArray[i]; // Each line of the time axis array.
    String[] splittedline = line.split(",");
    String splittedline_withcalendar = splittedline[1];

    String[] splittedline_withcalendar_butsplitted = splittedline_withcalendar.split(" ");
    String splittedline_onlyhour = splittedline_withcalendar_butsplitted[1];
    time_x[i] = splittedline_onlyhour;
}
```

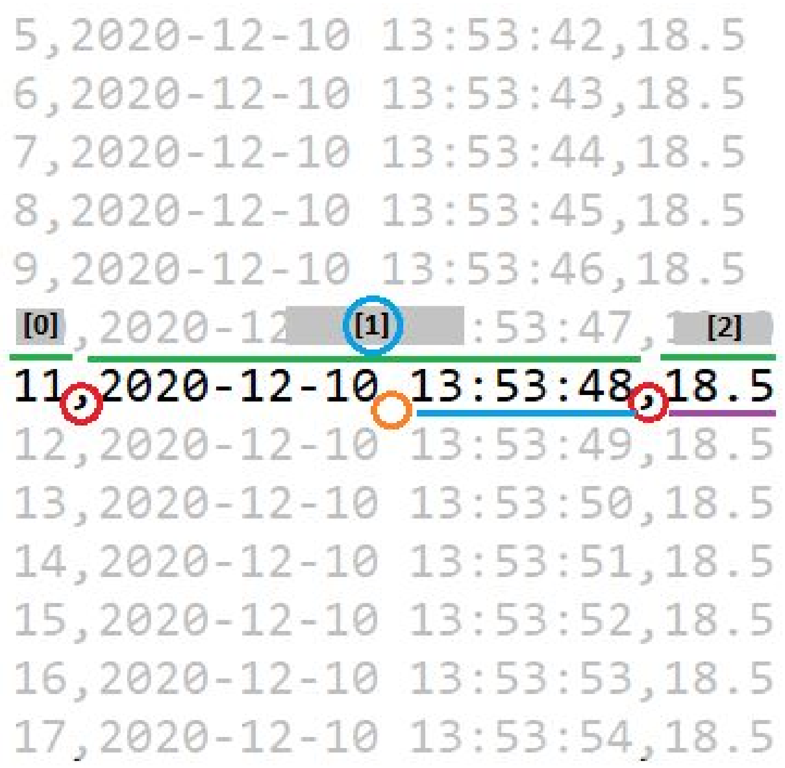
Now we split the date and time by the blank space in order to get the position of index=1 which corresponds to the time. Finally we save this time measurements in the time_x String array.

For the second string(values_y), we also split the line by the commas, but this time we take the position of index=2 which corresponds to the temperature measurements.

Finally we save each measurement in its corresponding position of the values_y String array.



```
for(int i = 0; i < stringArray.length; i++)
{
    String line = stringArray[i]; // Each line of the time axis array.
    String[] splittedline = line.split(",");
    String splittedline_withvalues = splittedline[2];
    values_y[i] = splittedline_withvalues;
}
```



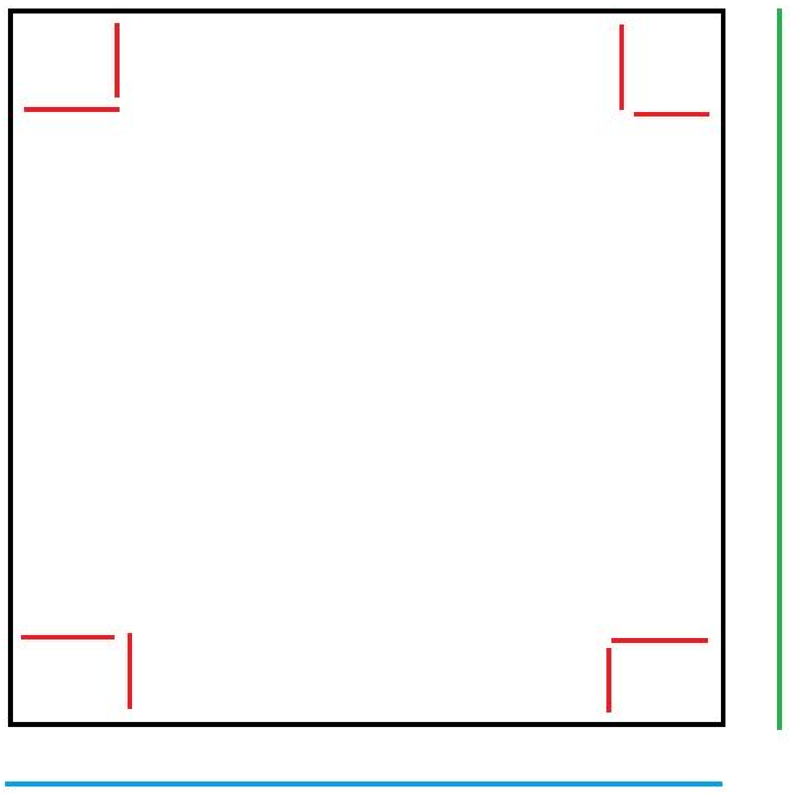
```
5,2020-12-10 13:53:42,18.5
6,2020-12-10 13:53:43,18.5
7,2020-12-10 13:53:44,18.5
8,2020-12-10 13:53:45,18.5
9,2020-12-10 13:53:46,18.5
10,2020-12-10 13:53:47,18.5
11,2020-12-10 13:53:48,18.5
12,2020-12-10 13:53:49,18.5
13,2020-12-10 13:53:50,18.5
14,2020-12-10 13:53:51,18.5
15,2020-12-10 13:53:52,18.5
16,2020-12-10 13:53:53,18.5
17,2020-12-10 13:53:54,18.5
```

From here, we start to draw in the program. For us it was important to make a flexible program that can easily adapt to a different resolution, or change the margin of the graph without breaking anything. So we have 3 main variables which are used in almost every pixel reference that we do in the program.

<u>"mar"</u> → The margin of the program, represented by the red lines.

"height" → The height of the window, getHeight().

"width" → The width of the window, getWidth().



```
int width = getWidth(); // Get res of the program.
int height = getHeight();
```

```
int mar = 60;
```

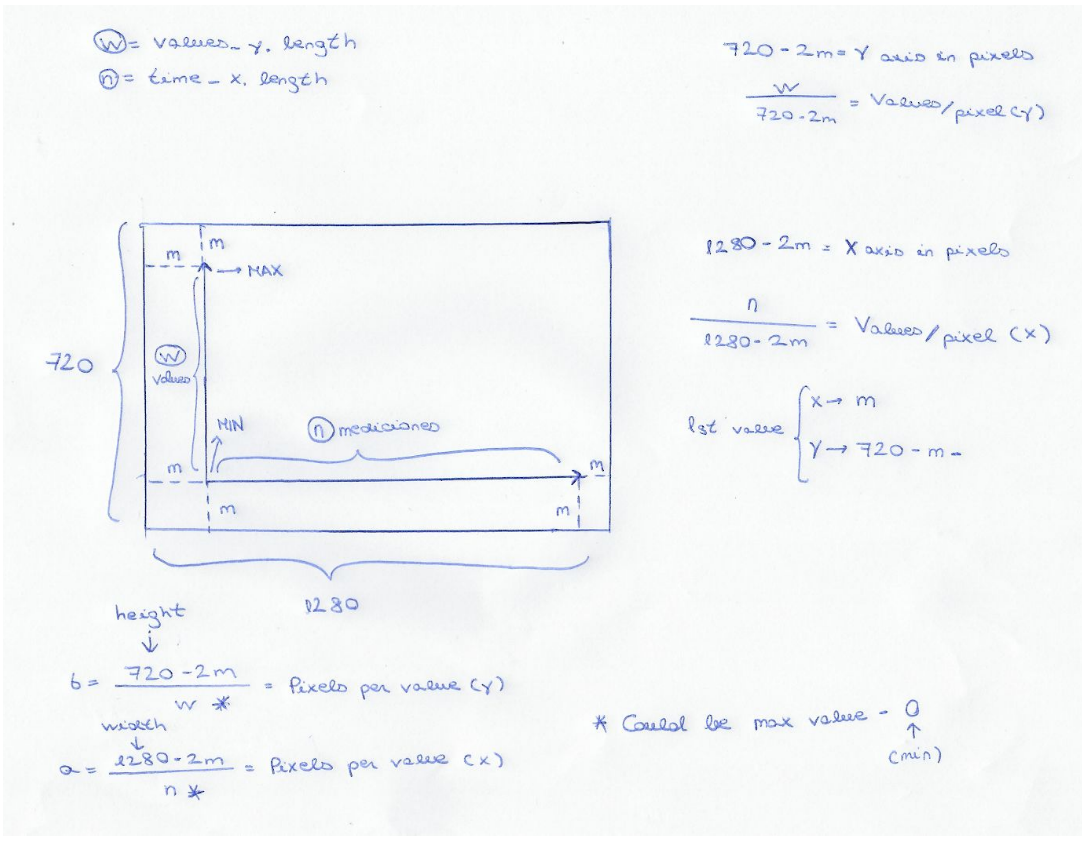
By doing this, we can represent the graph with the width, height, and margin that we want, not only for a standard one.

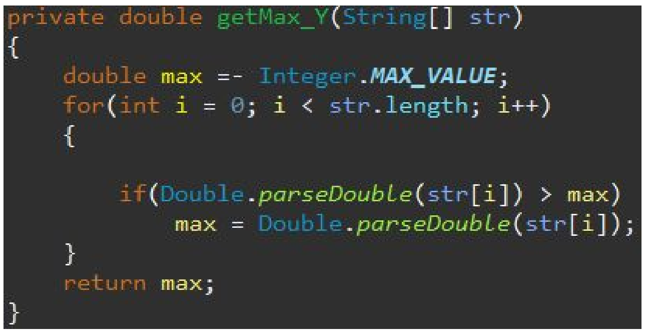
```
double a = (double) (width-2*mar)/values_y.length;
double b = (double) (height-2*mar)/getMax_Y(values_y);
```

We created two scales that will help us to represent the graph, one scale(a) is for the x axis and the scale(b) for the y axis.

For the a scale we divided the length of the x axis(in pixels) by the number of measurements of the file.

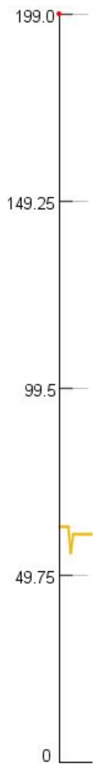
For the b scale first we tried dividing the height of the axis(in pixels) by the number of total measurements of the file, but in order to get a better view of the graph we decided to divide it by the maximum measurement of the file, thus creating the getMax_Y() method to get the highest temperature of the file, that way the highest temperature would be at the same height as the highest pixel of the y axis, providing us a better view of the graph.

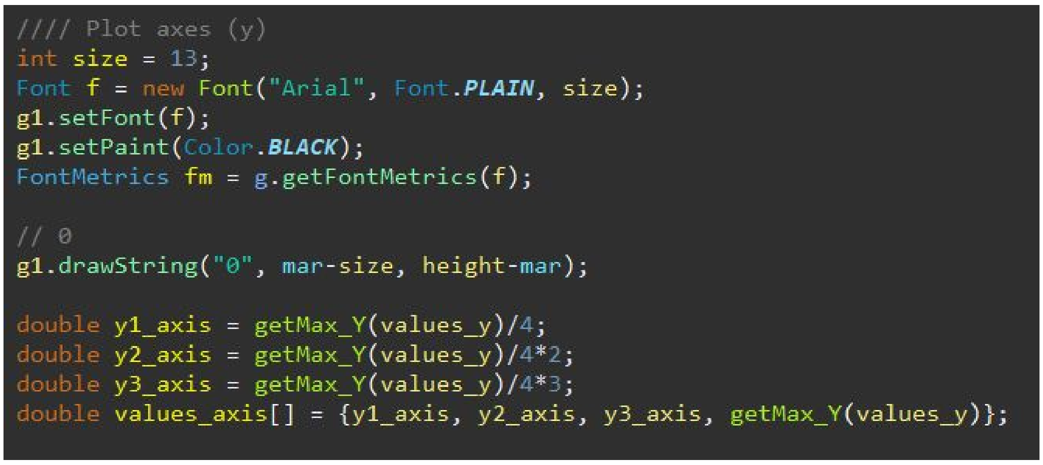

W = values_y. length
n = time_x. length
720 - 2m = Y axis in pixels
W / 720-2m = Values/pixel (y)
1280 - 2m = X axis in pixels
n / 1280-2m = Values/pixel (x)
1st value
X → m
Y → 720 - m
720
m
MAX
W values
MIN
n mediciones
1280
height
b = 720 - 2m / w * = Pixels per value (y)
width
a = 1280 - 2m / n * = Pixels per value (x)
* Could be max value - 0
(min)



```java
private double getMax_Y(String[] str)
{
    double max =- Integer.MAX_VALUE;
    for(int i = 0; i < str.length; i++)
    {

        if(Double.parseDouble(str[i]) > max)
            max = Double.parseDouble(str[i]);
    }
    return max;
}
```

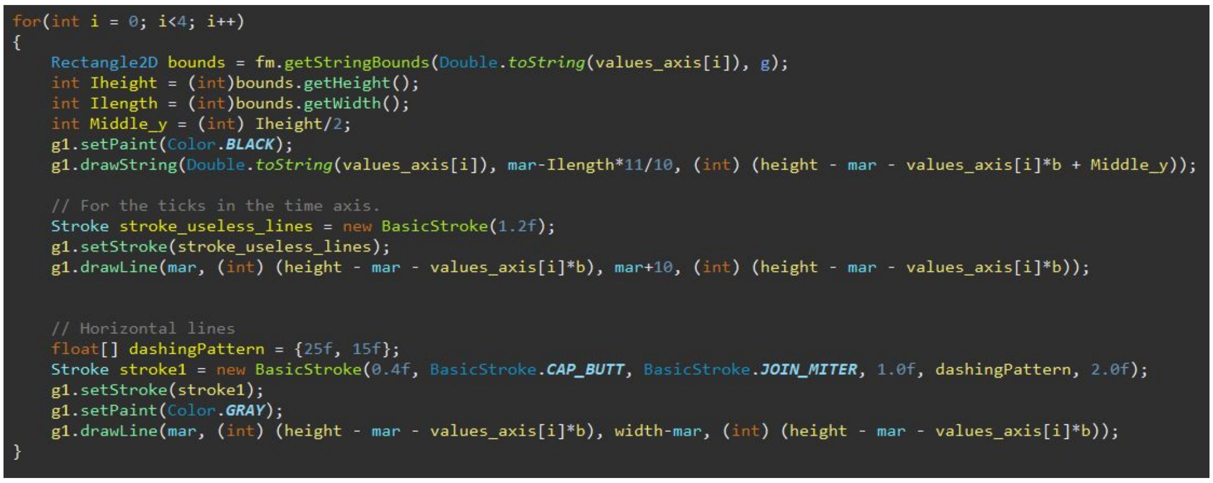
We also wanted to represent some marks on the axis to have a better understanding of the temperature.





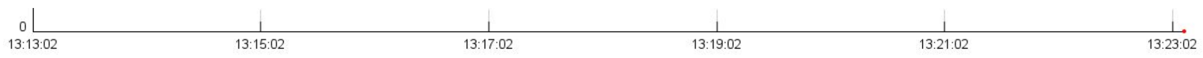
```
//// Plot axes (y)
int size = 13;
Font f = new Font("Arial", Font.PLAIN, size);
g1.setFont(f);
g1.setPaint(Color.BLACK);
FontMetrics fm = g.getFontMetrics(f);

// 0
g1.drawString("0", mar-size, height-mar);

double y1_axis = getMax_Y(values_y)/4;
double y2_axis = getMax_Y(values_y)/4*2;
double y3_axis = getMax_Y(values_y)/4*3;
double values_axis[] = {y1_axis, y2_axis, y3_axis, getMax_Y(values_y)};
```

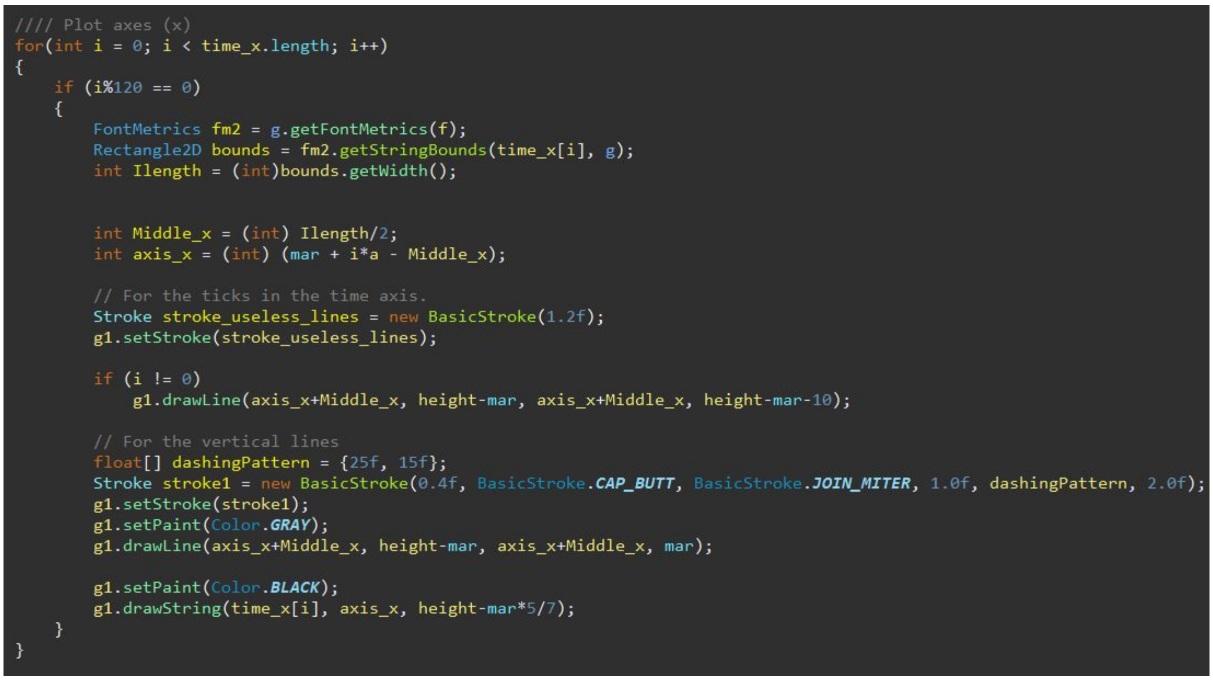
For this we had to think about the height in pixels of the text in order to center it with the small horizontal dashing lines that represent a value in the axis. We didn't want to overload it with values so we decided to plot four of them.



```
for(int i = 0; i<4; i++)
{
    Rectangle2D bounds = fm.getStringBounds(Double.toString(values_axis[i]), g);
    int Iheight = (int)bounds.getHeight();
    int Ilength = (int)bounds.getWidth();
    int Middle_y = (int) Iheight/2;
    g1.setPaint(Color.BLACK);
    g1.drawString(Double.toString(values_axis[i]), mar-Ilength*11/10, (int) (height - mar - values_axis[i]*b + Middle_y));

    // For the ticks in the time axis.
    Stroke stroke_useless_lines = new BasicStroke(1.2f);
    g1.setStroke(stroke_useless_lines);
    g1.drawLine(mar, (int) (height - mar - values_axis[i]*b), mar+10, (int) (height - mar - values_axis[i]*b));


    // Horizontal lines
    float[] dashingPattern = {25f, 15f};
    Stroke stroke1 = new BasicStroke(0.4f, BasicStroke.CAP_BUTT, BasicStroke.JOIN_MITER, 1.0f, dashingPattern, 2.0f);
    g1.setStroke(stroke1);
    g1.setPaint(Color.GRAY);
    g1.drawLine(mar, (int) (height - mar - values_axis[i]*b), width-mar, (int) (height - mar - values_axis[i]*b));
}
```

For the x axis we wanted to be more precise by representing the values every 2 minutes (120 seconds, 120 positions in the time_x[] array). We also had to keep in mind the length in pixels of the text string that contained the hour, minutes and seconds in order to center it with the small vertical dashing lines that are drawn every 2 minutes.

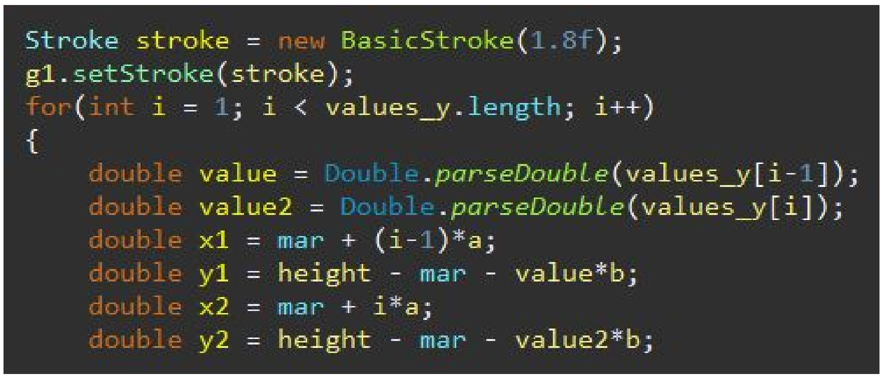




```java
//// Plot axes (x)
for(int i = 0; i < time_x.length; i++)
{
    if (i%120 == 0)
    {
        FontMetrics fm2 = g.getFontMetrics(f);
        Rectangle2D bounds = fm2.getStringBounds(time_x[i], g);
        int Ilength = (int)bounds.getWidth();


        int Middle_x = (int) Ilength/2;
        int axis_x = (int) (mar + i*a - Middle_x);

        // For the ticks in the time axis.
        Stroke stroke_useless_lines = new BasicStroke(1.2f);
        g1.setStroke(stroke_useless_lines);

        if (i != 0)
            g1.drawLine(axis_x+Middle_x, height-mar, axis_x+Middle_x, height-mar-10);

        // For the vertical lines
        float[] dashingPattern = {25f, 15f};
        Stroke stroke1 = new BasicStroke(0.4f, BasicStroke.CAP_BUTT, BasicStroke.JOIN_MITER, 1.0f, dashingPattern, 2.0f);
        g1.setStroke(stroke1);
        g1.setPaint(Color.GRAY);
        g1.drawLine(axis_x+Middle_x, height-mar, axis_x+Middle_x, mar);

        g1.setPaint(Color.BLACK);
        g1.drawString(time_x[i], axis_x, height-mar*5/7);
    }
}
```

Now we have to plot the graph. What we did is iterating the values starting with the second value of the file and plotting a line from this second value to the first value and so on, the lines are plotted from one value to the value before.This way we avoid out of index problems or getting some weird lines that shouldn't be plotted.

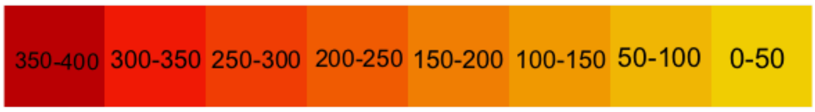
For each value, we had to get its position in pixels in order to draw the line from a point to the one before. As the (0,0) is in the top left corner, and the scale represents the number of pixels for each value, we can make the coordinates depending on the margin, height and its index, as it is shown in the image.



```java
Stroke stroke = new BasicStroke(1.8f);
g1.setStroke(stroke);
for(int i = 1; i < values_y.length; i++)
{
    double value = Double.parseDouble(values_y[i-1]);
    double value2 = Double.parseDouble(values_y[i]);
    double x1 = mar + (i-1)*a;
    double y1 = height - mar - value*b;
    double x2 = mar + i*a;
    double y2 = height - mar - value2*b;
```

```java
        g1.draw(new Line2D.Double(x1, y1, x2, y2));
        // g1.fill(new Ellipse2D.Double(x1, y1, 2, 2));
    }
```

As we said before, the graph is plotted as the maximum value is at the top of the graph, so we wanted to make it possible to compare the graphs of different files. We used a key of colours, so each temperature level is painted in a different colour.



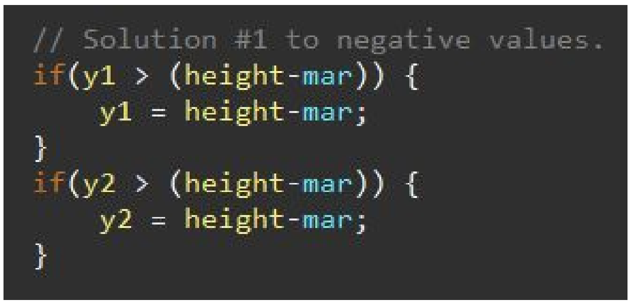
We also needed to plot the key in the window, so we fixed it in a little rectangle, so it adapts its resolution for every width and length of the program.

```java
// Colours in RGB
if (value<50) {
     g1.setPaint(new Color(240,205,3));
}
else if (value<100){
     g1.setPaint(new Color(240,182,3));
}
else if (value<150){
     g1.setPaint(new Color(240,153,3));
}
else if (value<200){
     g1.setPaint(new Color(240,126,3));
}
else if (value<250){
     g1.setPaint(new Color(240,90,3));
}
else if (value<300){
     g1.setPaint(new Color(240,61,3));
}
else if (value<350){
     g1.setPaint(new Color(240,25,3));
}
else if (value<400){
     g1.setPaint(new Color(186,0,3));
}
```

```java
int scale_key = 22;
int b2 = (int) (width/scale_key);
int a2 = (int) ((600/100) * b2);
```

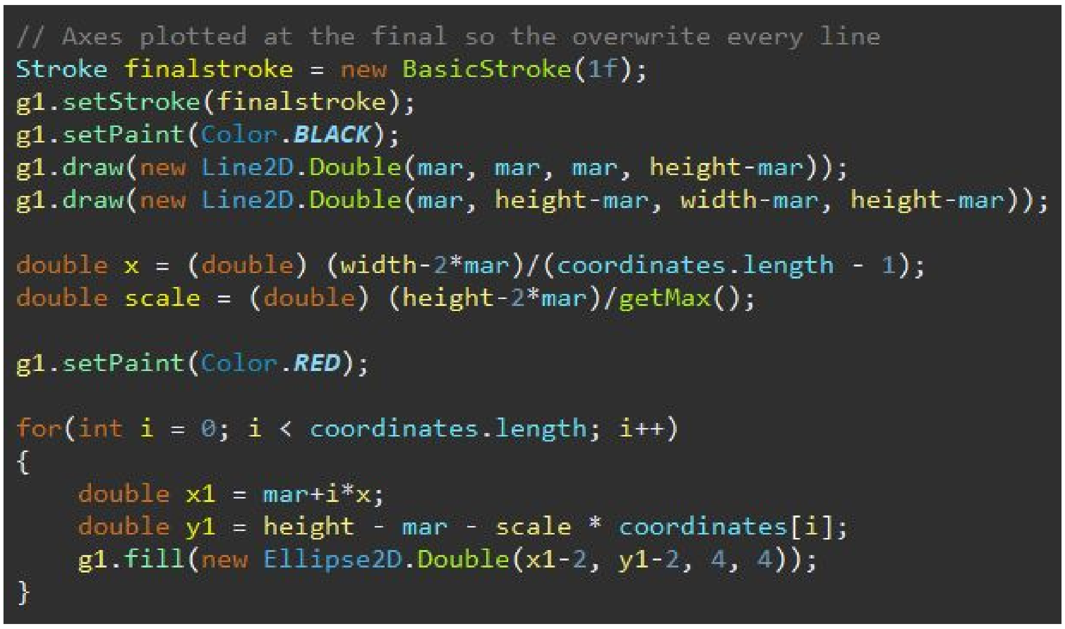
```java
// Plot key
Toolkit t =Toolkit.getDefaultToolkit();
Image key = t.getImage("key.png");
g.drawImage(key, mar, 0, a2, b2, this);
```

For the negative values we got because of unintentional measurements that appeared when the firelogger was turned on or off, when these negative values appeared we made them equal to zero in order to get a better looking graph and to avoid problems with the plotting of the axis.



```java
// Solution #1 to negative values.
if(y1 > (height-mar)) {
    y1 = height-mar;
}
if(y2 > (height-mar)) {
    y2 = height-mar;
}
```

We left the plotting of the axis for the end to make them appear on top of the other plotted elements.

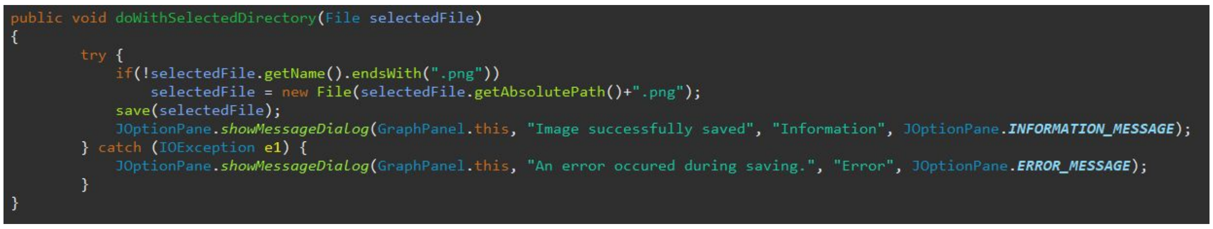


```java
// Axes plotted at the final so the overwrite every line
Stroke finalstroke = new BasicStroke(1f);
g1.setStroke(finalstroke);
g1.setPaint(Color.BLACK);
g1.draw(new Line2D.Double(mar, mar, mar, height-mar));
g1.draw(new Line2D.Double(mar, height-mar, width-mar, height-mar));

double x = (double) (width-2*mar)/(coordinates.length - 1);
double scale = (double) (height-2*mar)/getMax();

g1.setPaint(Color.RED);

for(int i = 0; i < coordinates.length; i++)
{
    double x1 = mar+i*x;
    double y1 = height - mar - scale * coordinates[i];
    g1.fill(new Ellipse2D.Double(x1-2, y1-2, 4, 4));
}
```

First we need to change the stroke to a normal stroke because we plotted some lines as dashing lines so we need to reformat that as well as the colour, changing it back to black. Next we draw the lines of the axis keeping in mind that the (0,0) point of the window is in the top left corner of the screen.

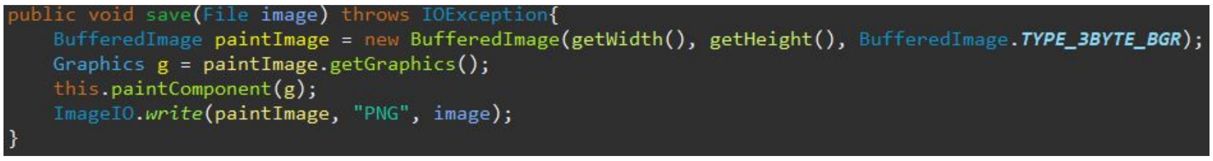
Finally we change the colour to red to plot some circles on the edge of the axis to see the end more clearly.

In the main program, if the “save” button was pressed, the method executed is the “doWithSelectedDirectory”, not the “save” one. This first method checks if the .png extension is in the path of the file to save, and shows the dialog message of the process. It also calls the method to save the image.



```
public void doWithSelectedDirectory(File selectedFile)
{
        try {
            if(!selectedFile.getName().endsWith(".png"))
                selectedFile = new File(selectedFile.getAbsolutePath()+".png");
            save(selectedFile);
            JOptionPane.showMessageDialog(GraphPanel.this, "Image successfully saved", "Information", JOptionPane.INFORMATION_MESSAGE);
        } catch (IOException e1) {
            JOptionPane.showMessageDialog(GraphPanel.this, "An error occured during saving.", "Error", JOptionPane.ERROR_MESSAGE);
        }
}
```

The method that saves the image, works in a very easy way. It executes again the “paintComponent” function that previously drew everything on the window, but this time it is “painted” in a PNG format.



```
public void save(File image) throws IOException{
    BufferedImage paintImage = new BufferedImage(getWidth(), getHeight(), BufferedImage.TYPE_3BYTE_BGR);
    Graphics g = paintImage.getGraphics();
    this.paintComponent(g);
    ImageIO.write(paintImage, "PNG", image);
}
```

Special thanks to stackoverflow & GitHub