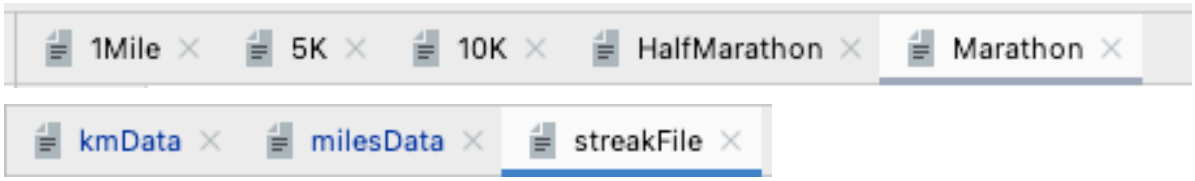
Criterion C: Development

The Runner App is a program that allows the client to input running data and displays it in a table. The program allows clients to sort between runs, toggle between units, and view PRs and streaks.

Classes:

- GUI.java generates the main UI of the program where the user will input data.
- History.java creates a frame that displays the run log, PR, and streaks.
- Data.java holds methods used in this program

Text Files:



- The first row of files holds specific distance runs (sorting)
- kmData.txt and milesData.txt hold all runs in both units.
- streakFile.txt holds an integer that represents the streak number.

Format of txt files

```
1   2022/02/07 21:03:32|00:06:30|1.61|00:04:02
2   2022/02/07 21:03:49|00:18:30|5.0|00:03:42
3   2022/02/07 21:03:52|00:18:30|10.0|00:01:51
4   2022/02/07 21:04:02|01:18:30|42.195|00:01:52
```

GUI.java

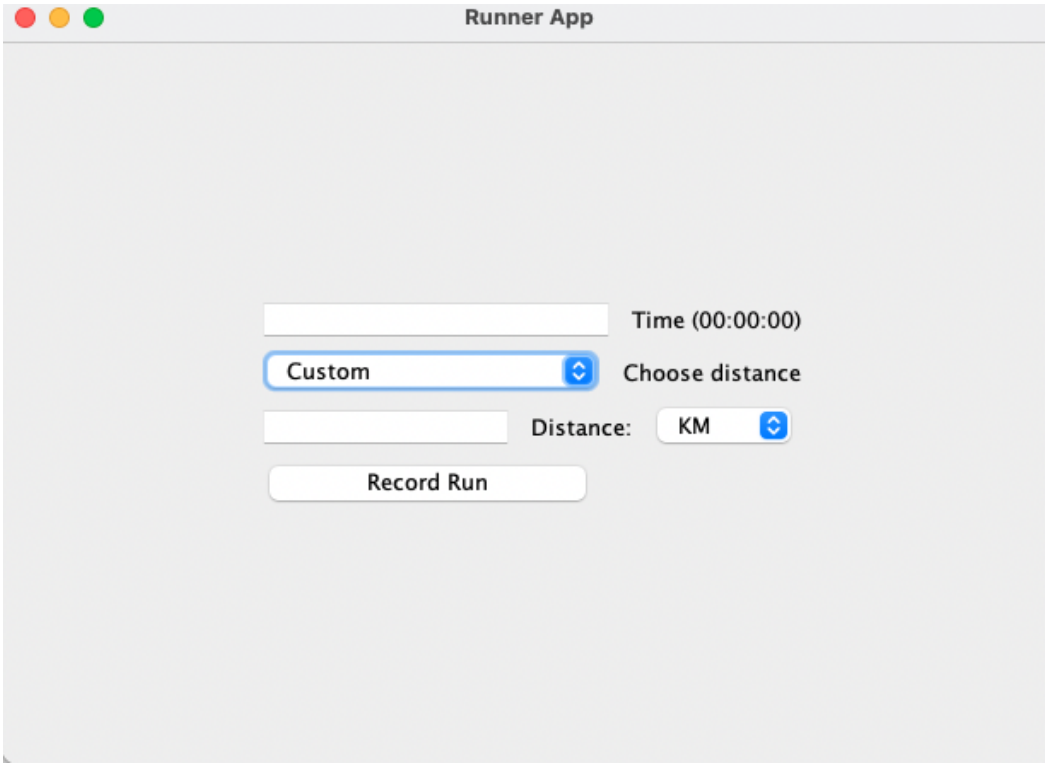


Figure 1: Main UI

Libraries imported:

```
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.*;
import java.time.*;
import java.time.format.DateTimeFormatter;
import java.util.Objects;
```

User interface:

The GUI of the program is created using javax.swing library.

The components of the UI are created with these instance variables:

```
private JPanel mainPanel;
private JTextField timeTextField;
private JLabel timeLabel;
private JButton addRunButton;
private JTextField distanceTextField;
private JComboBox unitsComboBox;
private JPanel EnterDistancePanel;
private JPanel selectionPanel;
private JComboBox selectionBox;
```

Figure 2: GUI.form

The UI is designed using GUI.form and instance variables (listed above) are created here.

The GUI class is a subclass of the imported JFrame class.

```
public class GUI extends JFrame {
```

Constructor

```
public GUI(String title) throws IOException { //constructor
    /////setup
    super(title);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setContentPane(mainPanel);
    this.pack();
    EnterDistancePanel.setVisible(false);
    /////
    selectionBox.addActionListener(new ActionListener() {...});
    unitsComboBox.addActionListener(new ActionListener() {...});
    if(Data.listCount() < 1 || (Data.streakCounter() == false && Data.sameDay() == false)){...}
    addRunButton.addActionListener(new ActionListener() {...});
}
```

The call of the super constructor passes the JFrame constructor the title of the frame which creates a functional JFrame.

this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); closes program if the JFrame window is closed.

this.setContentPane(mainPanel) sets the *mainPanel* instance variable as the main panel of the frame. The rest of the UI is built on top of the *mainPanel*.

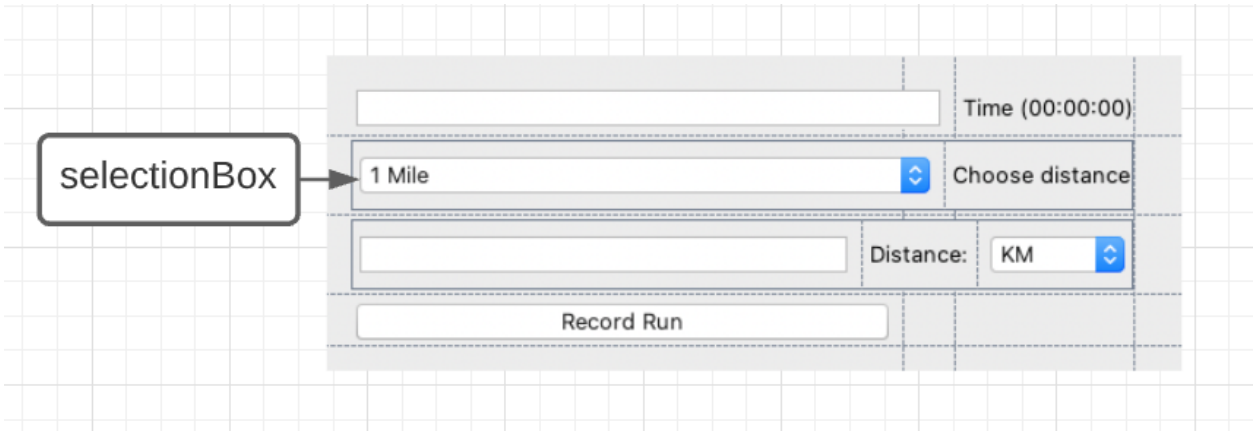
The addActionListeners in the constructor host the algorithmic code in the GUI class. The call of the addActionListener() takes the parameter ActionListener object which is responsible for handling action events when the user interacts with the component.

The *simple conditional* in the constructor is used to update the streak system and will be discussed later.

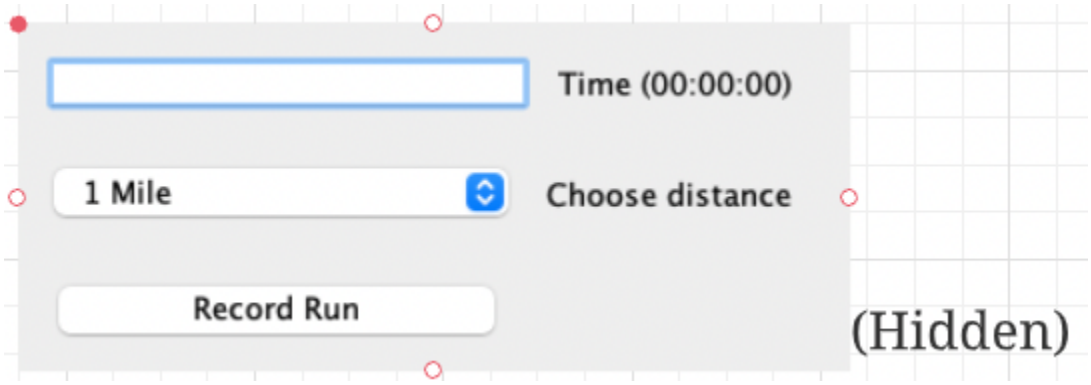
SelectionBox ActionListener

```
selectionBox.addActionListener(new ActionListener() { //distance dropbox
    @Override
    public void actionPerformed(ActionEvent e) {
        JComboBox temp = (JComboBox) e.getSource();
        String msg = (String) temp.getSelectedItem();
        if (msg.equals("1 Mile")) { //sets selection to user's request
            EnterDistancePanel.setVisible(false);
            selection = "Mile";
        } else if (msg.equals("5K")) {
            EnterDistancePanel.setVisible(false);
            selection = "Five";
        } else if (msg.equals("10K")) {
            EnterDistancePanel.setVisible(false);
            selection = "Ten";
        } else if (msg.equals("Half Marathon")) {
            EnterDistancePanel.setVisible(false);
            selection = "Half";
        } else if (msg.equals("Marathon")) {
            EnterDistancePanel.setVisible(false);
            selection = "full";
        } else if (msg.equals("Custom")) {
            EnterDistancePanel.setVisible(true);
            selection = "custom";
        }
    }
});
```

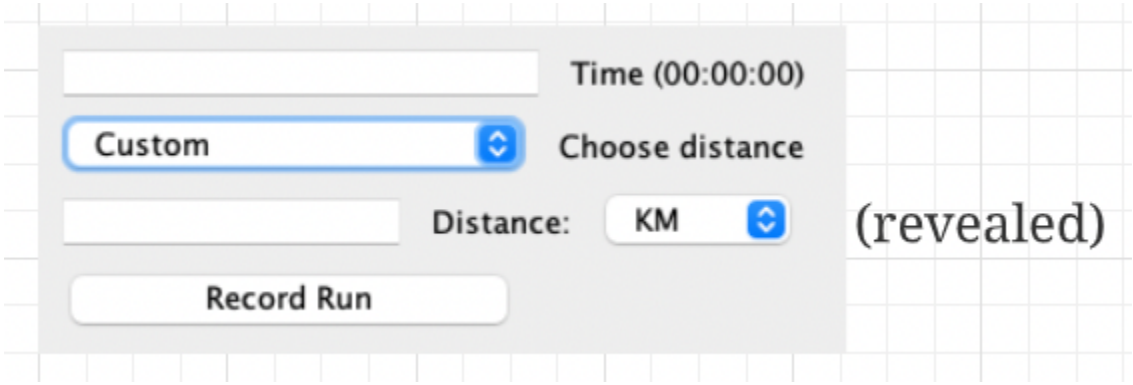
selectionBox is a JComboBox object that creates a dropbox in the UI that allows users to choose the distance they want to log.



The *conditionals* in the actionlistener modify the String *selection* instance variable which dictates which distance the user wants to record. For every selection except "custom," the method will set the JPanel *EnterDistancePanel* to false which will hide the custom distance textBox and the units dropbox.



If the user chooses the "custom" the JPanel *EnterDistancePanel* will become visible which reveals the custom distance textBox and the units dropbox.

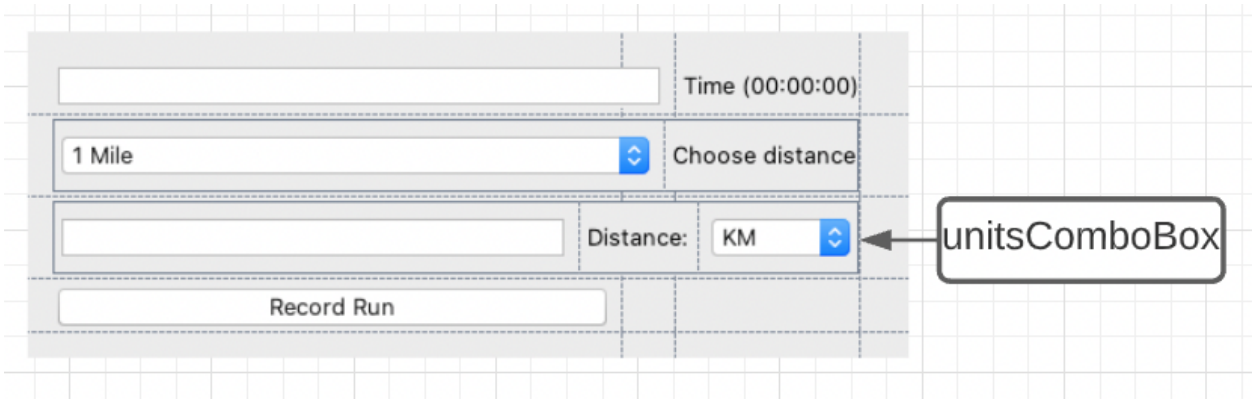


---

unitsComboBox ActionListener

```
unitsComboBox.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        JComboBox temp = (JComboBox) e.getSource();
        String msg = (String) temp.getSelectedItem();
        if (msg.equals("KM")) {
            status = true;
        } else {
            status = false;
        }
    }
});
```

unitsComboBox is a JComboBox object that is a dropbox that allows users to choose the units for their running distance.



The *conditionals* in the actionlistener modify the boolean *status* instance variable which dictates which units that the user wants to use. (true: KM, false: miles)

---

addRunButton ActionListener

```
addRunButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {...}
});
```

addRunButton is a JButton object that is a button that will perform the actionPerformed method when the user clicks it. The actionPerformed method is complex so the explanation will be broken up. The addRunButton adds the user inputted information into the correct files using *file i/o* and adjusts the integer in streakFile.txt that represents the streak number.

```
DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
LocalDateTime now = LocalDateTime.now();
String timeNow = dtf.format(now);
```

DateTimeFormatter formatas how the date/time is displayed.
LocalDateTime.now() retrieves the date/time of the specific instance.
timeNow is the String of date/time

Example: `2022/02/19 16:01:21`

```
String time = timeTextField.getText();
double distance = 0;
String pace = "";
```

String *time* holds the data that the user inputted as the duration of the run.
double *distance* and String *pace* are initialized to default values as it will be modified later in the method.

```
if (selection.equals("Mile")) {
    distance = 1;
    pace = Data.averagePace(distance, time);
    try {
        saveToFile( fileName: "1 Mile",  text: dtf.format(now) + "|" + time + "|" + distance + "|" + pace);
    } catch (IOException ioException) {
        ioException.printStackTrace();
    }
    try {
        saveToFile( fileName: "Miles",  text: dtf.format(now) + "|" + time + "|" + distance + "|" + pace);
    } catch (IOException ioException) {
        ioException.printStackTrace();
    }
    double distance2 = Data.milesToKm(distance);
    String pace2 = Data.averagePace(distance2, time);
    try {
        saveToFile( fileName: "KM",  text: dtf.format(now) + "|" + time + "|" + distance2 + "|" + pace2);
    } catch (IOException ioException) {
        ioException.printStackTrace();
    }
```

This *conditional* checks the value of the instance variable *selection* to see which files to add the collected information to. In this case, the selection is “mile”, so the distance variable is set to 1 and the pace is calculated using the averagePace() method (which will be explained later). The program then saves the data into 1Mile.txt which(sorted file) and the miles.txt & km.txt(all runs).

```
else {
  distance = Double.parseDouble(distanceTextField.getText());
  pace = Data.averagePace(distance, time);
  if (status == true) { //if selection is KM
      try {
          saveToFile( fileName: "KM",  text: dtf.format(now) + "|" + time + "|" + distance + "|" + pace);
      } catch (IOException ioException) {
          ioException.printStackTrace();
      }
      double distance2 = Data.kmToMiles(distance);
      String pace2 = Data.averagePace(distance2, time);

      try {
          saveToFile( fileName: "Miles",  text: dtf.format(now) + "|" + time + "|" + distance2 + "|" + pace2);
      } catch (IOException ioException) {
          ioException.printStackTrace();
      }
  } else {//if selection is Miles
      double distance2 = Data.milesToKm(distance);
      String pace2 = Data.averagePace(distance2, time);
      try {
          saveToFile( fileName: "Miles",  text: dtf.format(now) + "|" + time + "|" + distance + "|" + pace);
      } catch (IOException ioException) {
          ioException.printStackTrace();
      }
      try {
          saveToFile( fileName: "KM",  text: dtf.format(now) + "|" + time + "|" + distance2 + "|" + pace2);
      } catch (IOException ioException) {
          ioException.printStackTrace();
      }
  }
```

If the user chooses “custom”, the program will perform *a conditional within the original conditional (complex selection)*, which determines if the user chooses to input in kilometers or miles.

If the boolean instance variable *status* is true, this indicates that the user’s custom distance is in KM and the program will save the data in kmData.txt and convert distance/pace into miles and store it in milesData.txt. Vice versa for *status* equals false.

```
tableFrame.dispose();// close old table update new one
try {
    tableFrame = new history( title: "Run Log");//make the history jframe.
} catch (FileNotFoundException fileNotFoundException) {
    fileNotFoundException.printStackTrace();
}
```

Finally, the program will dispose of the tableFrame (the run log frame) and create a new table which will refresh/update the data displayed.

---

history.java

Libraries imported:

```
import javax.swing.*;
import javax.swing.table.DefaultTableModel;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.*;
```

User Interface:

Instance variables that make up the run log GUI.

```
private JPanel tablePanel;
private JTable table;
private JComboBox comboBox1;
private JComboBox SortComboBox;
private JPanel togglePanel;
private JPanel prLabelPanel;
private JLabel icon;
private JLabel streakLabel;
private ImageIcon crown;
private ImageIcon fire;
```

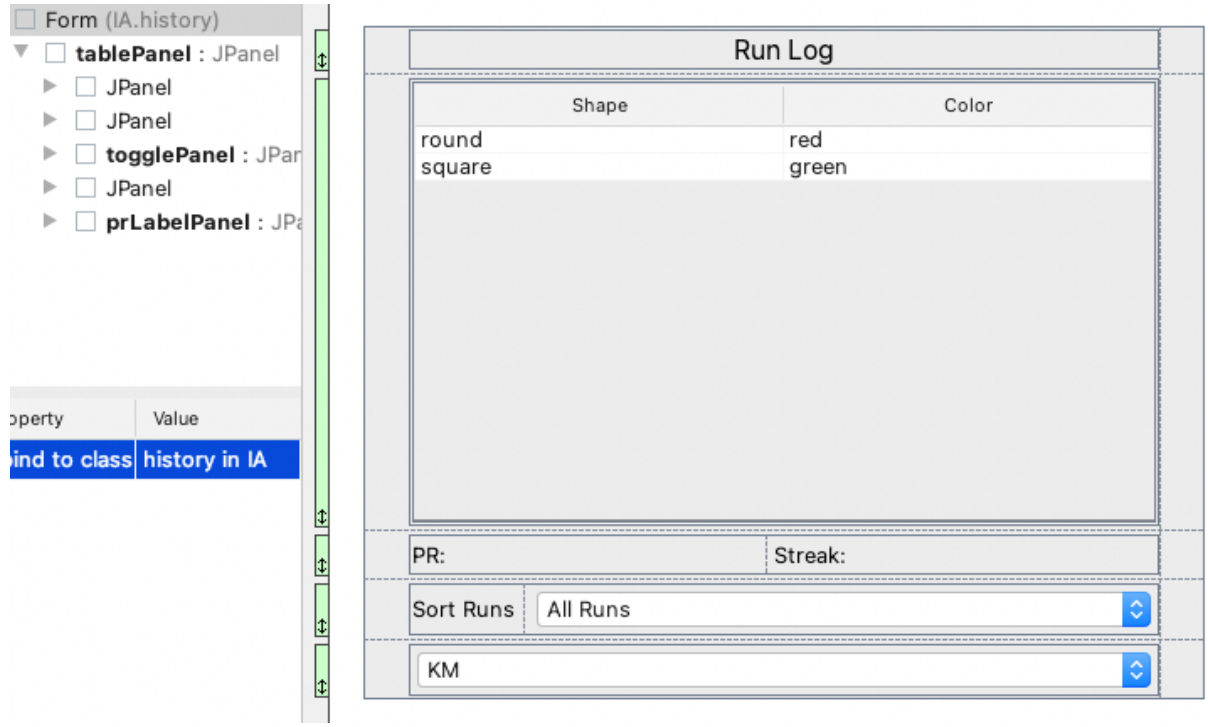


Figure 3: history.form

The UI is designed using history.form and instance variables are created here.

2D Object arrays used to store data in each text file so it can be displayed on a table.

```
Object[][] data = fileArr(new File( pathname: "/U
Object[][] data2 = fileArr(new File( pathname: "/
Object[][] mile = fileArr(new File( pathname: "/U
Object[][] five = fileArr(new File( pathname: "/U
Object[][] ten = fileArr(new File( pathname: "/Us
Object[][] half = fileArr(new File( pathname: "/U
Object[][] marathon = fileArr(new File( pathname
''''
```

The method *fileArr* is called which returns a 2D array given parameter File Object.

```
public static Object [][] fileArr(File file) throws FileNotFoundException {
    Scanner sc = new Scanner(file);
    int length = 0;
    while(sc.hasNextLine()) {
        sc.nextLine();
        length++;
    }
    Object[][] temp = new Object[length][4];
    Scanner sc2 = new Scanner(file);
    for(int i = 0; i<temp.length; i++){
        temp[i] = sc2.nextLine().split( regex: "\\|");
    }
    return temp;
}
```

While loop is used to get the length of the file. Then, a *for loop* is used to set each cell of the 2D array with the data of one recorded run. sc2.nextLine().split() returns an array stored with data that is split using the param key.

Figure 4: Table methods

```
private void createTableKm() { //creates km table
    table.setModel(new DefaultTableModel(
            data,
            new String[]{"Date/time", "Duration", "Distance(KM)", "Pace mins/km"}
    ));
}
private void createTableMile() { //creates mile table
    table.setModel(new DefaultTableModel(
            data2,
            new String[]{"Date/time", "Duration", "Distance(Mi)", "Pace mins/mi"}
    ));
}
```

Figure 4 fills the JTable *table* instance variable with data. It then takes a *2D array* that consists of the data to be displayed and a *1D array* that contains the table header.

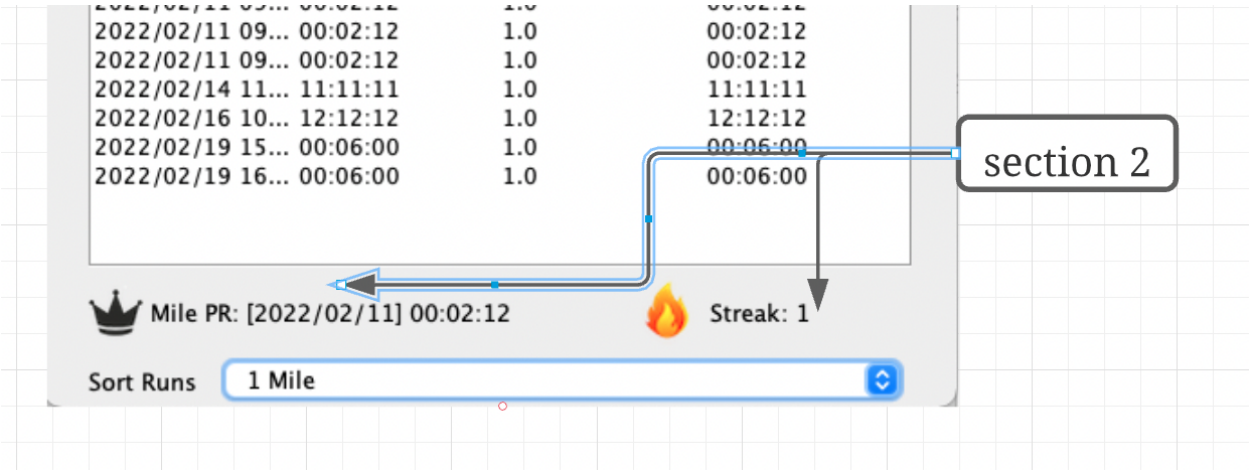
Constructor

```
public history(String title) throws FileNotFoundException {
    /////setup
    super(title);
    this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    this.setContentPane(tablePanel);
    this.setVisible(true);
    this.pack();
    ////

    //section 2
    crown = new ImageIcon(this.getClass().getResource( name: "crown.png"));//crown png
    icon.setIcon(crown);//crown icon set
    icon.setText("Distance PR: "+ Data.findLongestDate()+
            " ("+Data.kmToMiles(Double.parseDouble(Data.finLongestDistanceKm()))+"Miles) "
            + "(" +Data.finLongestDistanceKm()+ "KM)");//display longest distance
    fire = new ImageIcon(this.getClass().getResource( name: "streakImg.png"));//fire png
    streakLabel.setIcon(fire);//streak icon
    streakLabel.setText("Streak: " + getStreak());//streak number display
    //

    if (status = true) { //COMPLEXITY 4
        createTableKm();
    } else {
        createTableMile();
    }
    comboBox1.addActionListener(new ActionListener() {...});
    SortComboBox.addActionListener(new ActionListener() {...});
}
```

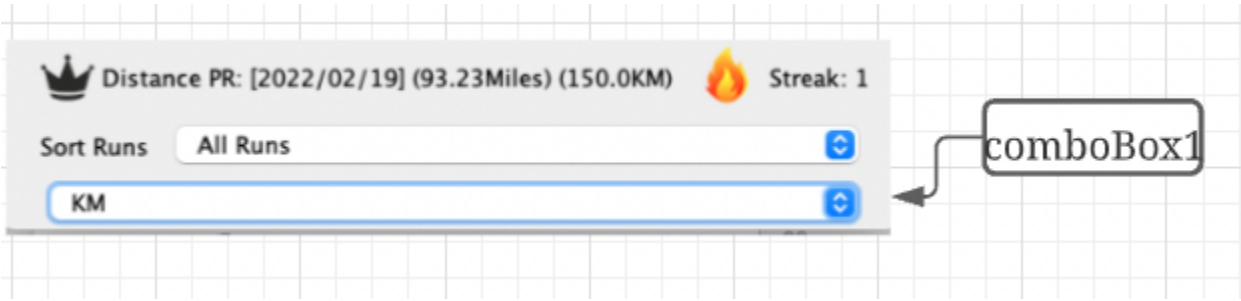
The //setup// portion of the history constructor is the same GUI constructor.
//Section 2 of the is responsible for creating this portion of the run log. (methods used will be discussed later)



```java
comboBox1.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        JComboBox temp = (JComboBox) e.getSource();
        String msg = (String) temp.getSelectedItem();
        if (msg.equals("KM")) {
            status = true;
            createTableKm(); //display km table
        } else {
            status = false;
            createTableMile();//display miles table
        }

    }
});
```

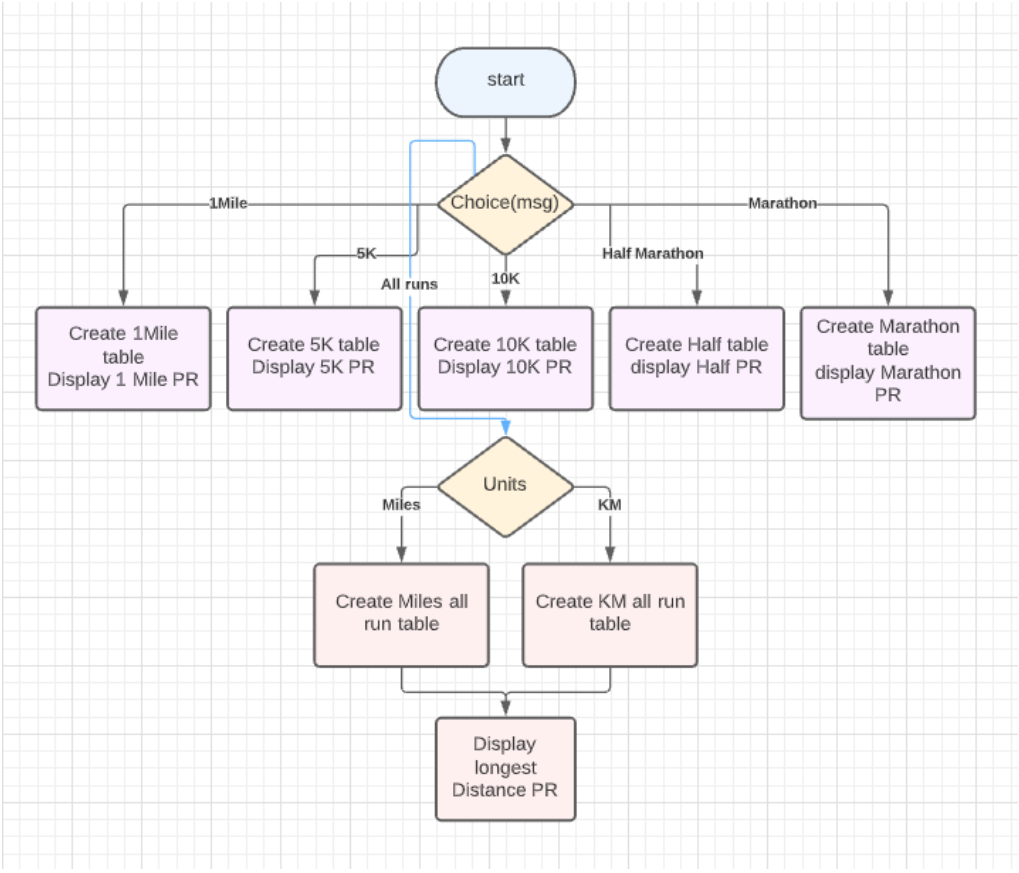
JComboBox *comboBox1* is a dropbox with two selections (KM/Miles) with the *conditional* if user selects "KM" status(of the table) is set to true(km) and will call createTableKM() method. Vice versa when the user selects "Miles."



```
SortComboBox.addActionListener(new ActionListener() { //
    @Override
    public void actionPerformed(ActionEvent e) {
        JComboBox temp = (JComboBox) e.getSource();
        String msg = (String) temp.getSelectedItem();
        if(msg.equals("1 Mile")){ //display 1 mile table
            togglePanel.setVisible(false);
            table.setModel(new DefaultTableModel(
                    mile,
                    new String[]{"Date/time", "Duration"
            ));
            try {
                icon.setText("Mile PR: " + Data.findPR(
            } catch (IOException ioException) {
                ioException.printStackTrace();
            }
        }
        else if(msg.equals("5K")){ //display 5k table
```

JComboBox *SortComboBox* lets the user sort their runs.
Conditionals in the actionPerformed control what's being displayed.

Figure 5: SortComboBox actionlistener flowchart



Tables are created using the method discussed in Figure 4. *(2D&1D array complexity)*

Data.java

Libraries imported:

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.math.BigDecimal;
import java.math.RoundingMode;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.*;
```

Data.java contains *user defined methods* that are called in the other two classes.

Figure 6: *Searching* for the longest distance in the file.

```
public static String finLongestDistanceKm() throws FileNotFoundException
    String longestDistance = "";
    File file = new File( pathname: "/Users/william/IdeaProjects/School/IB
    Scanner sc = new Scanner(file);
    double longest = 0;
    while(sc.hasNextLine()){
        String [] tempArr = sc.nextLine().split( regex: "\\|");
        double temp = Double.parseDouble(tempArr[2]);
        if(temp>longest){
            longest = temp;
        }
    }
    longestDistance = Double.toString(longest);
    return longestDistance;
}
```

This linear *searching* algorithm (discussed in Criterion B) returns the longest distance recorded in the file. The *while loop* traverses every line of the file and compares the distances. Finally, it returns the longest distances as a string.

List of all User Defined Methods in Data.java

Method Signature	Description
formatTime(int hours, int mins, int secs)	Accepts *parameters* of 3 integers (hours, minutes, seconds) and *returns* a string of the formatted time ("hh:mm:ss")
averagePace(double distance, String time)	Accepts a double for distance and String for duration in the *parameter* and *returns* average pace.
kmToMiles(double km)	Returns miles given km input in *parameter*
milesToKm(double miles)	Returns km given miles input *parameter*
listCount()	Counts the number of lines in a file(All run file)
streakCounter()	Checks if the most recent run is the day before today, if so then streak is not broken and *returns* true
sameDay()	Check if the most recent entry is on the same day as the one just entered. If it is, then *returns* true
saveToFile(String fileName, String text)	Save imputed text to the requested file given the *parameters* String fileName, and String text.
incrementStreak(String choice)	Increments number in the streakFile.txt by the amount indicated by the string passed in the *parameter*
findPR(String fileName)	Returns fastest time in given file
findLongestDistanceKm()	Find the longest distance ever recorded and return the longest distance in KM. Uses *Searching* to loop through the file.
findLongestDate()	Returns the date of the longest run. Uses *Searching* to loop through the file.
timeToSeconds(String time)	Returns time in seconds given time in

Word Count: 868