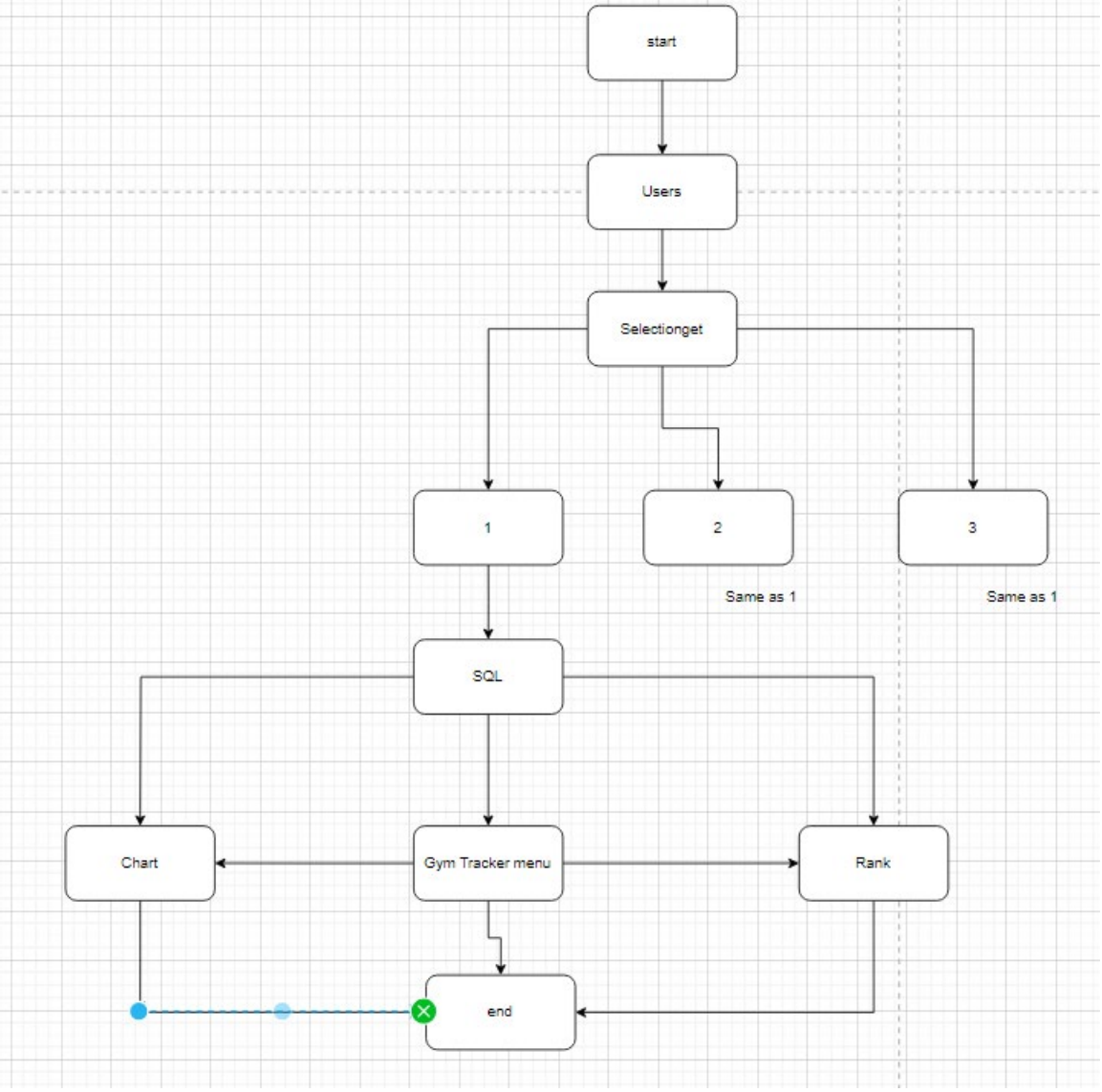
DESIGN REPORT

Gym Tracker was designed to help user store data about their workout. Using the SQL data system, it stores many users' data, help them sort them by data and performance. In addition, it demonstrates their progress using charts drawn in gosu, Finally, it sorts the users data and create a leaderboard.

Program Flowchart



1. GymTracker_functions

- The custom program does not allow $ global variable, therefore I used modules to communicate between the classes as I need to save the data and print them from TextInput class and Home class

  Here's an explanation of how the `DataManager` module works:

1. `@selection` is an instance variable within the module. It's an array used to store selected values. This can be thought of as a way to temporarily store some value for later use.

2. `@shared_data` is another instance variable used to store data that needs to be shared across different parts of the application.

3. `add_to_data(value)` is a method that adds the given `value` to the `@shared_data` array.

4. `get_data` is a method that returns the contents of the `@shared_data` array.

5. `delete_data` is a method that clears the contents of the `@shared_data` array, effectively resetting it.

6. `add_selection(value)` is a method that adds the given `value` to the `@selection` array.

7. `delete_selection` is a method that clears the contents of the `@selection` array.

8. `get_selection` is a method that retrieves the first value from the `@selection` array. This implies that `@selection` is used to store a single selected value at a time.

By using this module, you can achieve data sharing and management in a more controlled and encapsulated manner compared to using global variables like `$variables`. Global variables can lead to unintended side effects and make it harder to reason about the state of your application.

```
11
12
13  module DataManager
14    @selection=[]
15
16    @shared_data = []
17
18    def self.add_to_data(value)
19      @shared_data << value
20    end
21
22    def self.get_data
23      @shared_data
24    end
25
26    def self.delete_data
27      @shared_data = []
28    end
29
30    def self.add_selection(value)
31      @selection << value
32    end
33
34    def self.delete_selection
35      @selection = []
36    end
37
38    def self.get_selection
39      @selection[0]
40    end
41  end
```

- Merge sort to sort data
  This is a customized merge sort that compares the sum of the 1 and 2 index integer

The provided code contains two functions: `merge_sort_2d` and `merge`. These functions together implement a modified version of the merge sort algorithm to sort a 2D array based on the sum of specific columns in each row.

1. `merge_sort_2d(arr)` function:

This function is the entry point for the merge sort algorithm. It takes a 2D array `arr` as input and aims to return a sorted version of the input array.

- `num_elements`: This variable stores the number of rows in the array.

- Base case: If there's only one element or no elements in the array, it returns the array as it is (base case for recursion).

- `middle`: Calculates the midpoint of the array.

- `left_half`: Splits the input array into the left half, containing elements from index 0 up to (but not including) the middle.

- `right_half`: Splits the input array into the right half, containing elements from the middle index to the end.

- Recursively calls `merge_sort_2d` on the left and right halves.

- Calls the `merge` function to combine and sort the left and right halves.

- Returns the sorted and merged array.

2. `merge(left, right)` function:

This function takes two sorted arrays (`left` and `right`) and merges them while maintaining the sorted order based on a specific criterion (the sum of values in columns 1 and 2).

- `sorted_arr`: This array will hold the merged and sorted result.

- `left_index` and `right_index`: These variables keep track of the current index being considered in the left and right arrays, respectively.

- The `while` loop continues until either the `left_index` reaches the end of the left array or the `right_index` reaches the end of the right array.

- `left_sum` and `right_sum`: Calculate the sum of values in columns 1 and 2 for the current row in the left and right arrays.

- Compares the sums. If the sum from the left array is less than or equal to the sum from the right array, the row from the left array is added to `sorted_arr`, and `left_index` is incremented. Otherwise, the row from the right array is added to `sorted_arr`, and `right_index` is incremented.

- This loop continues until one of the arrays is fully processed.

- Finally, any remaining elements from either the left or right array are added to `sorted_arr`.

In summary, the `merge_sort_2d` function recursively divides the input 2D array into smaller halves and then uses the `merge` function to merge and sort these halves based on the sum of values in specific columns. This approach results in a sorted 2D array where rows are ordered based on their column sums.

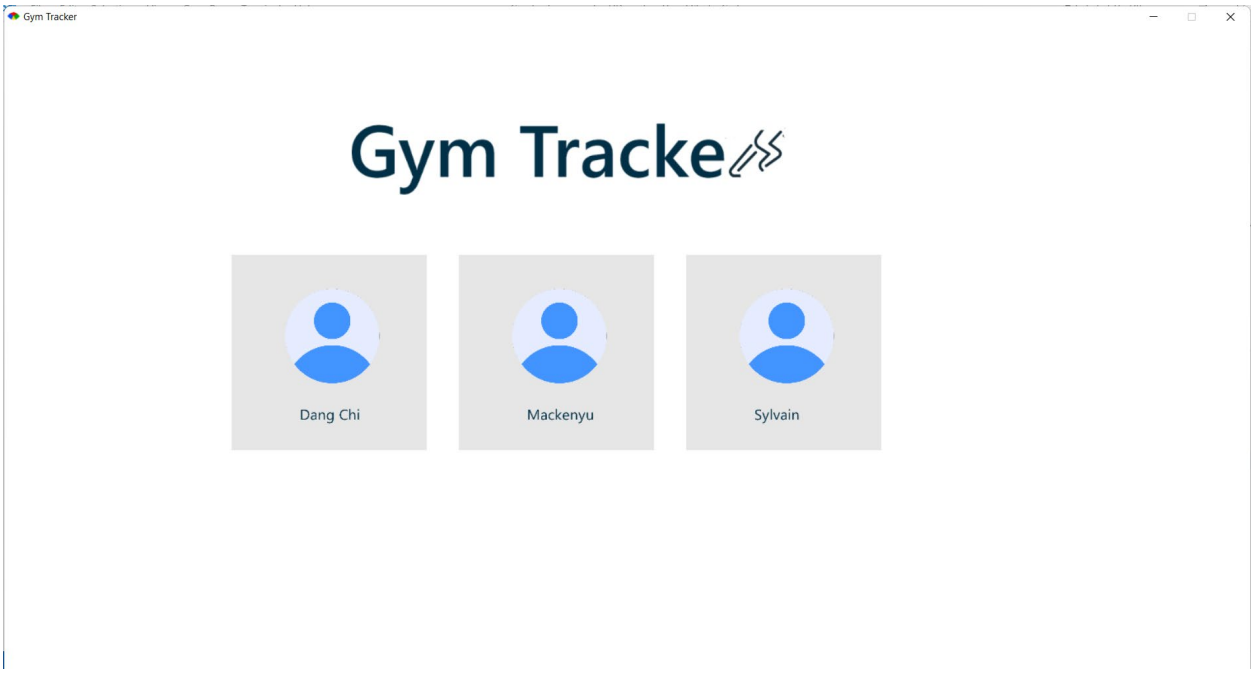
```ruby
def merge_sort_2d(arr)
  num_elements = arr.length

  return arr if num_elements <= 1

  middle = num_elements / 2
  left_half = arr[0...middle]
  right_half = arr[middle..-1]

  sorted_left = merge_sort_2d(left_half)
  sorted_right = merge_sort_2d(right_half)

  merge(sorted_left, sorted_right)
end

def merge(left, right)
  sorted_arr = []
  left_index, right_index = 0, 0

  while left_index < left.length && right_index < right.length
    left_sum = left[left_index][1].to_i + left[left_index][2].to_i
    right_sum = right[right_index][1].to_i + right[right_index][2].to_i

    if left_sum <= right_sum
      sorted_arr << left[left_index]
      left_index += 1
    else
      sorted_arr << right[right_index]
      right_index += 1
    end
  end
```

- Constant for days and time:

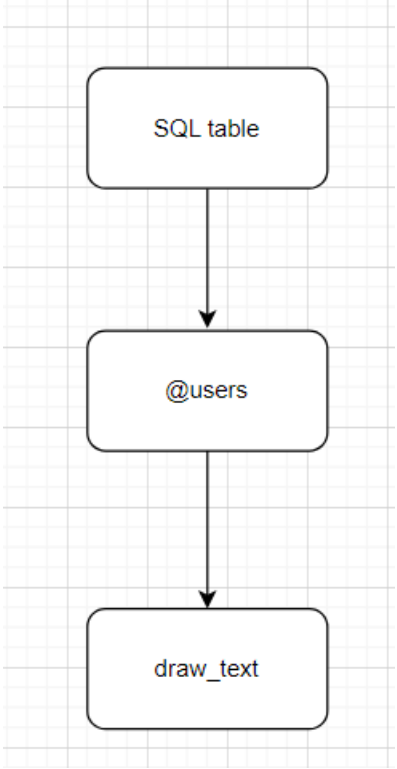
```ruby
DAY = ['Sunday', 'Monday', 'Tuesday', ' Wednesday', 'Thursday','Friday','Saturday']
MONTH = [ 'January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October', 'November', 'December']
```

-

2. GymTracker_users



The menu open up with users and their data
The users names are printed by selecting the users_name in the table

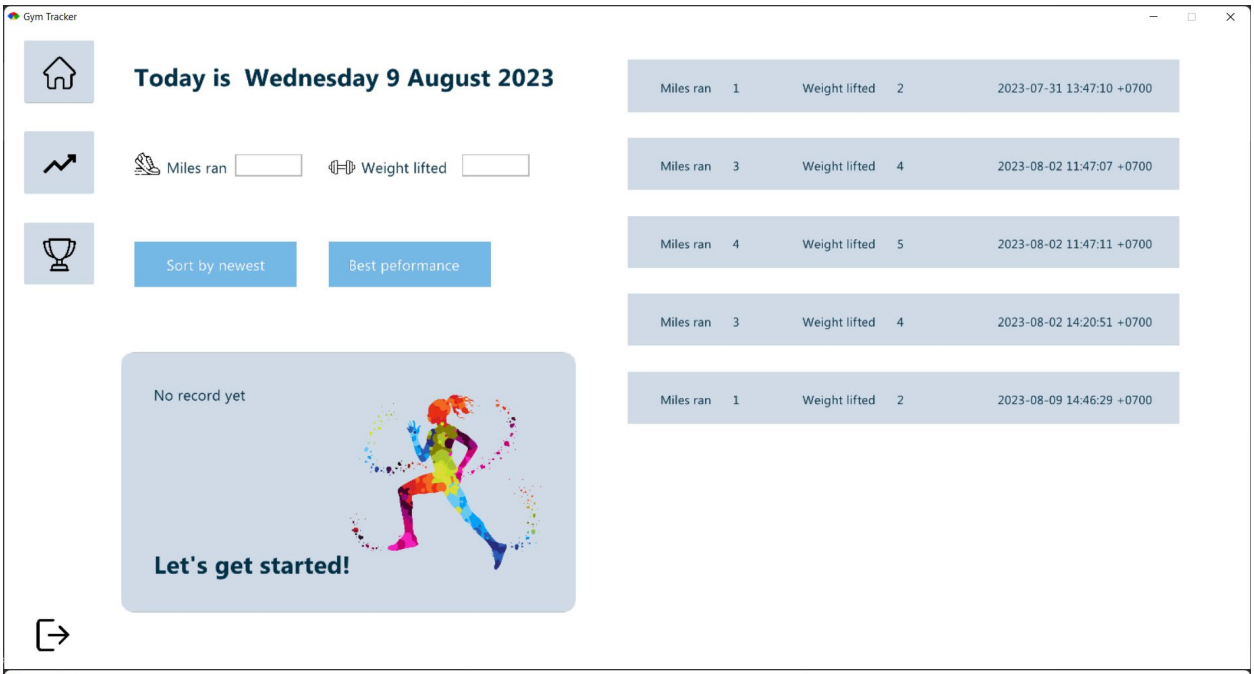


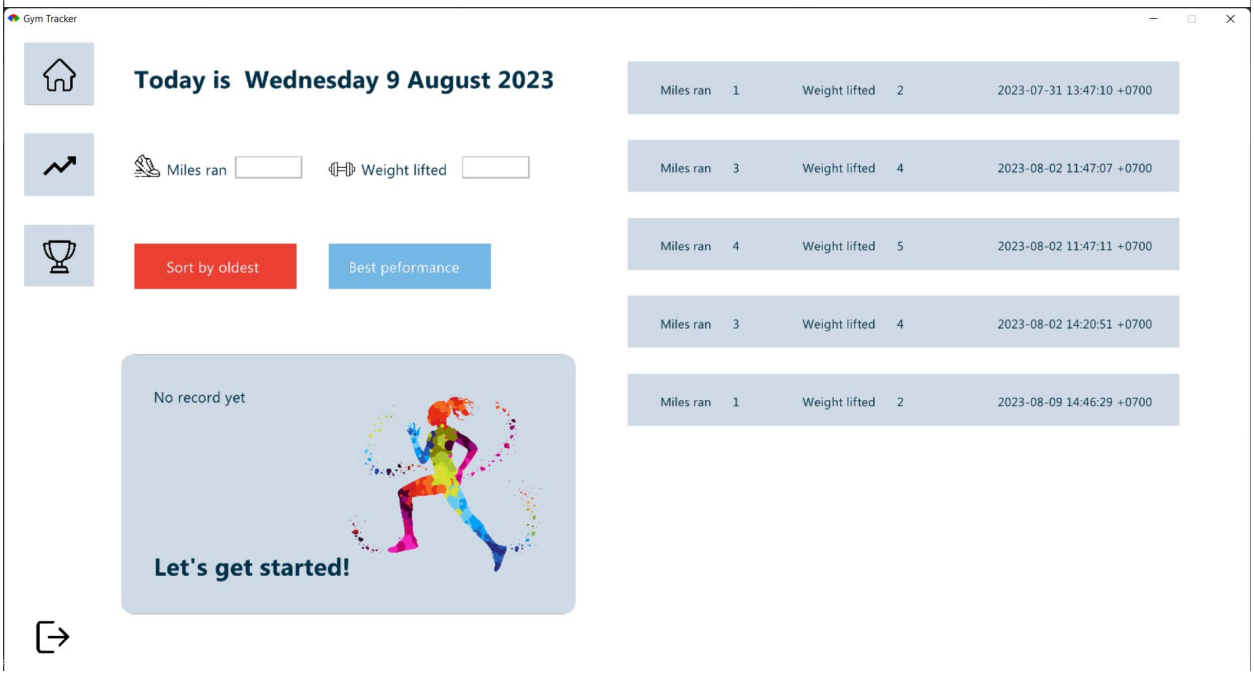
When you choose a user, it will assign the user data for them to use it to print later on

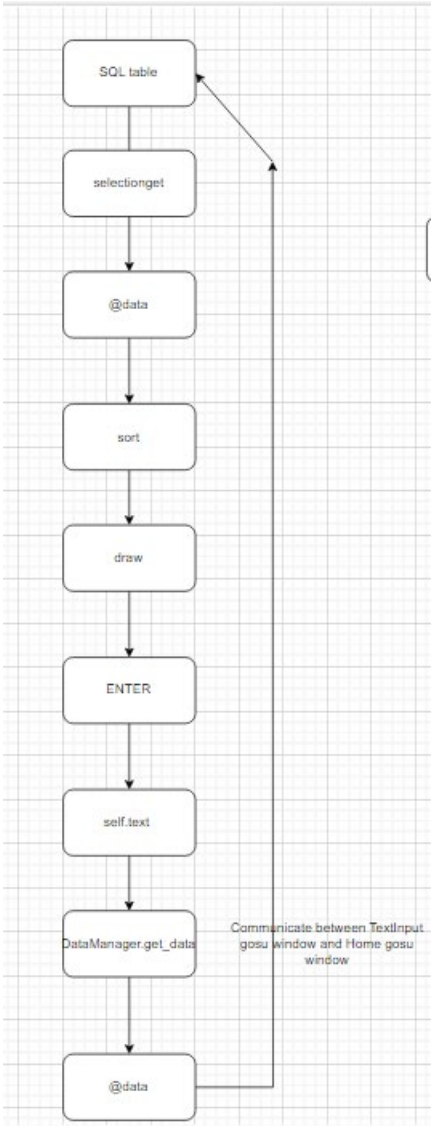
```
    case id
when Gosu::MsLeft
        case area_clicked
            when 1
                # $select = 1
                DataManager.add_selection(1)
                require './GymTracker.rb'
                close
                Home.new.show if __FILE__ == $0
            when 2
                # $select = 2
                DataManager.add_selection(2)
                require './GymTracker.rb'
                close
                Home.new.show
            when 3
                # $select = 3
                DataManager.add_selection(3)
                require './GymTracker.rb'
                close
                Home.new.show

        end
    end
end
```

Selection = []
adding selection
The selection will be the data that connects the users_id to the right user data

3. GymTracker


Gym Tracker
Today is Wednesday 9 August 2023
Miles ran
Weight lifted
Sort by newest
Best peformance
No record yet
Let's get started!
Miles ran 1 Weight lifted 2 2023-07-31 13:47:10 +0700
Miles ran 3 Weight lifted 4 2023-08-02 11:47:07 +0700
Miles ran 4 Weight lifted 5 2023-08-02 11:47:11 +0700
Miles ran 3 Weight lifted 4 2023-08-02 14:20:51 +0700
Miles ran 1 Weight lifted 2 2023-08-09 14:46:29 +0700


Gym Tracker
Today is Wednesday 9 August 2023
Miles ran
Weight lifted
Sort by oldest
Best peformance
No record yet
Let's get started!
Miles ran 1 Weight lifted 2 2023-07-31 13:47:10 +0700
Miles ran 3 Weight lifted 4 2023-08-02 11:47:07 +0700
Miles ran 4 Weight lifted 5 2023-08-02 11:47:11 +0700
Miles ran 3 Weight lifted 4 2023-08-02 14:20:51 +0700
Miles ran 1 Weight lifted 2 2023-08-09 14:46:29 +0700



The work flow of how data is stored and drawn in sql tables, sql is initialized only once when the program is opened or upon require"./", therefore, for real-time updates on gosu, I programmed it so it inserts data into sql when RETURN key is pressed

```ruby
def sql
  begin
    @db = SQLite3::Database.new "data_and_users.db"

    # Create gym_users table
    @db.execute <<-SQL
      CREATE TABLE IF NOT EXISTS gym_users (
        users_id INTEGER PRIMARY KEY,
        users_name TEXT NOT NULL
      );
    SQL

    # Insert data into gym_users table
    @db.execute <<-SQL
      INSERT INTO gym_users (users_name)
      VALUES
        ('Dang Chi'),
        ('Mackenyu'),
        ('Sylvain');
    SQL

    # Create gym_data table (parent table)
    @db.execute <<-SQL
      CREATE TABLE IF NOT EXISTS gym_data (
        data_id INTEGER PRIMARY KEY,
        miles INTEGER NOT NULL,
        weight INTEGER NOT NULL,
        time INTEGER NOT NULL,
        users_id INTEGER NOT NULL,
        FOREIGN KEY (users_id) REFERENCES gym_users (users_id)
      );
    SQL


    # Fetch data from gym_data table for a specific user (using $select vari
```

1. The method `sql` is defined. This method seems to be responsible for setting up a connection to an SQLite3 database and performing various database operations.

2. Inside the `begin` block, a connection to the SQLite3 database named "data_and_users.db" is established using `SQLite3::Database.new`.

3. The code creates a table named `gym_users` if it doesn't already exist. This table has columns `users_id` (an auto-incrementing primary key) and `users_name` (a non-null text field).

4. Data for gym users is inserted into the `gym_users` table. Three users with names 'Dang Chi', 'Mackenyu', and 'Sylvain' are inserted.

5. Another table named `gym_data` is created (if it doesn't exist) to store exercise data. This table has columns `data_id`, `miles`, `weight`, `time`, and `users_id`. The `users_id` column is a foreign key referencing the `users_id` column in the `gym_users` table, establishing a relationship between users and their exercise data.

6. The code seems to fetch data from the `gym_data` table for a specific user. The query uses a placeholder (`?`) and references the `DataManager.get_selection` method to retrieve the selection criteria.

7. The fetched data is stored in the `@data` array. The exact structure of `@data` isn't provided in the code snippet, but it's assumed to be an array of rows from the fetched result set.

8. The code is wrapped in a `begin` block, which is followed by a `rescue` block. This block catches and handles exceptions of type `SQLite3::Exception` that might occur during database operations. Any caught exception is printed to the console.

9. The `ensure` block is used to close the database connection after all the database operations are done. However, this part of the code is commented out with the comment, "If the whole application is going to exit...". This implies that the code might be part of a larger application where the database connection handling could be managed elsewhere.

- When clicking on a user, it will set your selection so it can find the data from the user

```
# Fetch data from gym_data table for a specific user (using $select variable)
@db.execute("SELECT data_id, miles, weight, time, users_id FROM gym_data WHERE users_id = ?", DataManager.get_selection) do |row|
  @data << row
end
```

- Here is the textinput to save the data using clone

```
def save_to_data
  data = self.text.chomp.to_i.clone
  DataManager.add_to_data(data)
  self.text = ''
end
```

-

```
if self.text_input
  @text_fields[0].save_to_data
  @text_fields[1].save_to_data
  data = DataManager.get_data
```

-
-
- Saving the text input into an array to then be inserted into the sql table and drawn onto the gosu interface
- Greeted by “Today is …” using Time.now.day using module to accurate present the time

```
# include Gosu::TextInput
def draw_home
  #menu background
  draw_rect(0, 0, SCREEN_W, SCREEN_H, MENU_COLOR, ZOrder::BACKGROUND, mode = :default)


  #texts
  BOLD_FONT.draw_markup("Today is #{DAY[Time.now.wday.to_i]} #{Time.now.day} #{MONTH[Time.now.month.to_i-1]} 2023 ", 200, 50, ZOrder::UI, 1.0, 1.0, FONT_COLOR)
  REG_FONT.draw_text("Miles ran ", 250, 200, ZOrder::UI, 1.0, 1.0, FONT_COLOR)
  REG_FONT.draw_text("Weight lifted", 550, 200, ZOrder::UI, 1.0, 1.0, FONT_COLOR)
  @weight.draw_rot(500,195,10,0,0,0,img_size(@shoes.width,40), img_size(@shoes.height,40), Gosu::Color::WHITE)
  @shoes.draw_rot(200,190,10,0,0,0,img_size(@weight.width,40), img_size(@weight.height,40), Gosu::Color::WHITE)
  @text_fields.each { |tf| tf.draw(0) }
```

- You can sort by newest and oldest (cloned the array and used reverse)

```
  draw_sort_by
  #menu
  time=@data.clone


  if @sort_by_best == nil && @sort_by == nil
    draw_history(time)
  end


 # miles ran and weight lifted
  if @sort_by


    draw_history(time)
  elsif @sort_by == false
    draw_history(time.reverse)
  end

  sorted_array = merge_sort_2d(@data).clone
  if @sort_by_best
    # @sorted_by_sum_of_first_and_second_elements = merge_sort_2d(@data).reverse
    draw_sort_by_best(sorted_array)
  elsif @sort_by_best== false
    # @sorted_by_sum_of_first_and_second_elements = merge_sort_2d(@data)
    draw_sort_by_best(sorted_array.reverse)
  end
  #drawing the status of workout
  draw_status
end
```

-

- You can sort by best and worse performance(customized merge sort to compare the sum of the two intergers)

```ruby
def merge_sort_2d(arr)
  num_elements = arr.length

  return arr if num_elements <= 1

  middle = num_elements / 2
  left_half = arr[0...middle]
  right_half = arr[middle..-1]

  sorted_left = merge_sort_2d(left_half)
  sorted_right = merge_sort_2d(right_half)

  merge(sorted_left, sorted_right)
end

def merge(left, right)
  sorted_arr = []
  left_index, right_index = 0, 0

  while left_index < left.length && right_index < right.length
    left_sum = left[left_index][1].to_i + left[left_index][2].to_i
    right_sum = right[right_index][1].to_i + right[right_index][2].to_i

    if left_sum <= right_sum
      sorted_arr << left[left_index]
      left_index += 1
    else
      sorted_arr << right[right_index]
      right_index += 1
    end
  end
```

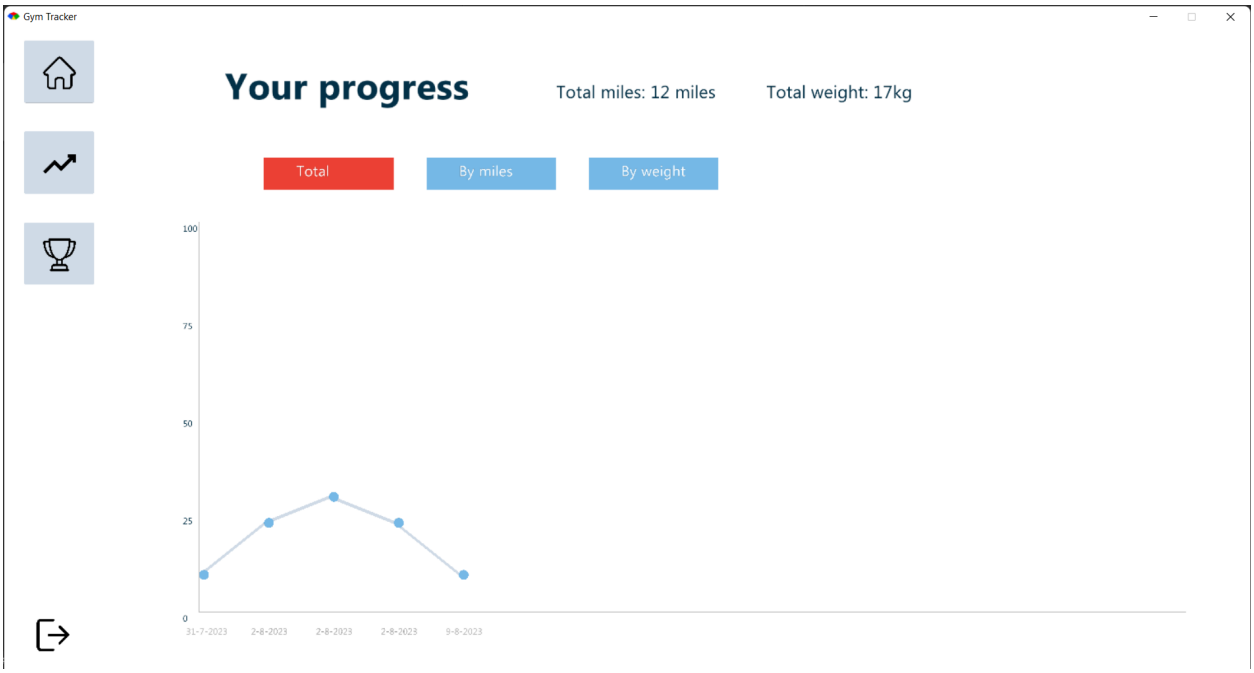
-
- A status bar will update whether or not you've done any exercise today (using @data)

```ruby
RECT.draw_rot(box_x,box_y, ZOrder::BACKGROUND,0,0,0,img_size(RECT.width,700), img_size(RECT.height,400), LIGHT_BLUE)
if data.empty?
 @run_status.draw_rot(box_x+340,box_y+40,ZOrder::BACKGROUND,0,0,0,img_size(@run_status.width,320), img_size(@run_status.height,320), BOX_COLOR)
 REG_FONT.draw_text("No record yet",box_x+50,box_y+50, ZOrder::UI, 1.0, 1.0, FONT_COLOR)
 BOLD_FONT.draw_text("Let's get started!",box_x+50,box_y+300, ZOrder::UI, 1.0, 1.0, FONT_COLOR)
else
 @trophy_status.draw_rot(box_x+350,box_y+60,ZOrder::BACKGROUND,0,0,0,img_size(@run_status.width,230), img_size(@run_status.height,230), BOX_COLOR)
 REG_FONT.draw_text("Today you did",box_x+50,box_y+50, ZOrder::UI, 1.0, 1.0, FONT_COLOR)
 BOLD_FONT.draw_text("#{miles.sum} miles \nand #{weight.sum} weight!",box_x+50,box_y+250, ZOrder::UI, 1.0, 1.0, FONT_COLOR)
end
```

-

- Scrollable gym data records using mouse_x + 15, add if statements in order for the records to not overflow through the window's screen

4. Chart screen



- Your progress. The total miles and weight is drawn using each loop and miles weight arrays.

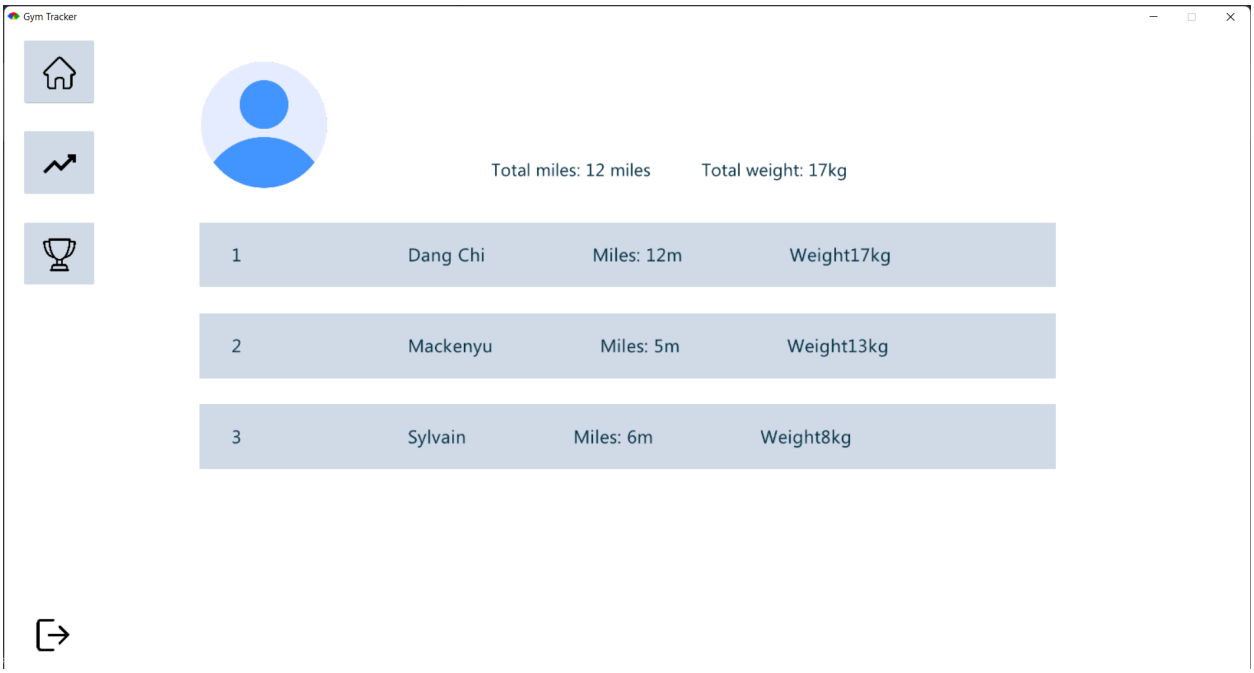
```
def draw_total
    miles =[]
    weight=[]

    i=0
    time = @data.size
    time.times do
      miles << @data[i][1].clone.to_i
      weight<< @data[i][2].clone.to_i

      i+=1
    end

    # REG_FONT.draw_text("#{@data}", 850, 80, ZOrder::UI, 1.2, 1.2, FONT_COLOR)
    REG_FONT.draw_text("Total miles: #{miles.sum} miles          Total weight: #{weight.sum}kg", 850, 80, ZOrder::UI, 1.2, 1.2, FONT_CO
end
```

  - Drawing the chart: I wanted to draw a line graph, however, it would interfere with eachothers as there are always 1 less line drawn for every dot in the chart, Therefore, I used a loop to draw the circle twice and the line once

```
@multiplier = 20
n=0
i=0
  if @miles_chart ==  true
    line.times do
      y1 = @data[n][1].to_i * @multiplier
      y3 = @data[n+1][1].to_i * @multiplier

      draw_quad 302+i, line_y  - y1, LIGHT_BLUE, 302+i, line_y  - y1 + 5, LIGHT_BLUE, 402+i, line_y  - y3, LIGHT_BLUE, 402+i, line_y  - y
      @circle.draw_rot(300+i,dot_y - y1,ZOrder::BACKGROUND,0,0,0,img_size(@circle.width,15), img_size(@circle.height,15), BLUE)
      @circle.draw_rot(400+i,dot_y - y3,ZOrder::BACKGROUND,0,0,0,img_size(@circle.width,15), img_size(@circle.height,15), BLUE)
      REG_FONT.draw_text "#{@date[n][2]}-#{@date[n][1]}-#{@date[n][0]}", 280+i, SCREEN_H - 80, ZOrder::UI, 0.6, 0.6, OUTLINE_COLOR
      REG_FONT.draw_text "#{@date[n+1][2]}-#{@date[n+1][1]}-#{@date[n+1][0]}", 380+i, SCREEN_H - 80, ZOrder::UI, 0.6, 0.6, OUTLINE_COLOR
      i+=100
      n+=1
      end


  elsif @weight_chart == true
```

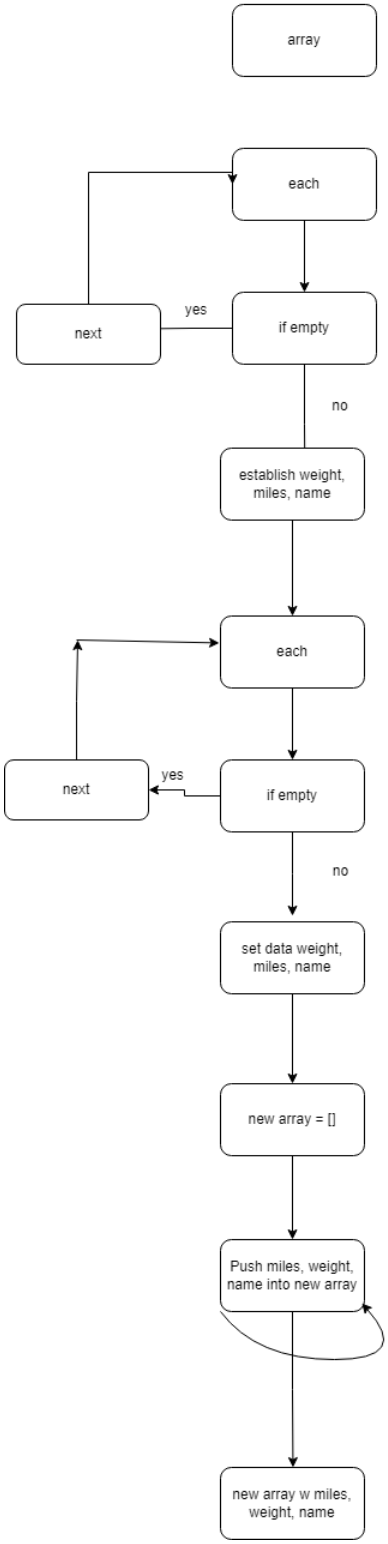
  - You can sort by Total, Miles or Weight (using Boolean statements and if)

5. Rank screen



- To do the rankings of the accounts, I inner join the tables by users_id and push the into Array.(@users.size){[]} where @users.size is the amount of users name and {[]} makes every element inside the array an array.
- Then I loop each to replace the arrays in the array with their weight and miles sum for easier comparison

Flowchart:


array
each
yes
next
if empty
no
establish weight, miles, name
each
next
yes
if empty
no
set data weight, miles, name
new array = []
Push miles, weight, name into new array
new array w miles, weight, name

```ruby
#m did not work because it was a local variable, it did not save
    def users_ranked
        n=0
        @your_rank= DataManager.get_selection

        @wow = @users_data.clone
        @wow.each do
          next if @wow[n].empty? # Skip empty arrays

          total_miles = 0
          total_weight = 0
          user_name = nil
          user_id = nil

          @wow[n].each do |data|
            next if !data.is_a?(Array) # Skip non-array elements
            user_id ||= data[5].to_i
            total_miles += data[1].to_i
            total_weight += data[2].to_i
            user_name ||= data[6].to_s

          end


          @wow[n]=[]
          @wow[n].push(user_id,total_miles,total_weight,user_name)
          n +=1
        end

        @leader_board=merge_sort_2d(@wow).reverse
        @leader_board.each_with_index  do |m, index|
        Gosu.draw_rect 300 , 300+140*index, SCREEN_W-600, 100, LIGHT_BLUE, ZOrder::BACKGROUND
        SMALL_FONT.draw_text("#{index+1} "+" "*30+"#{m[3]}"+" "*20+"Miles: #{m[1]}m"+" "*20+"Weight#{m[2]}kg", 350, 330+140*index, ZOrder::UI,

        end
```

-
- Then I proceed to sort it using merge sort to produce the leaderboard

↓

Then it is used with merge sort to draw the leader board from most miles and weight to least