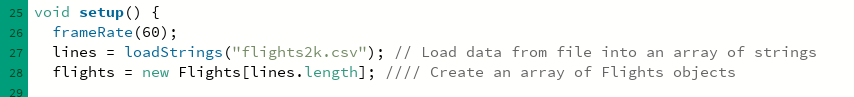
**CSU11013 - Team 31 Report**

With a separate minute’s file on our repo, this report will focus on documenting our different source files and their functionality within our completed project.

**Main**

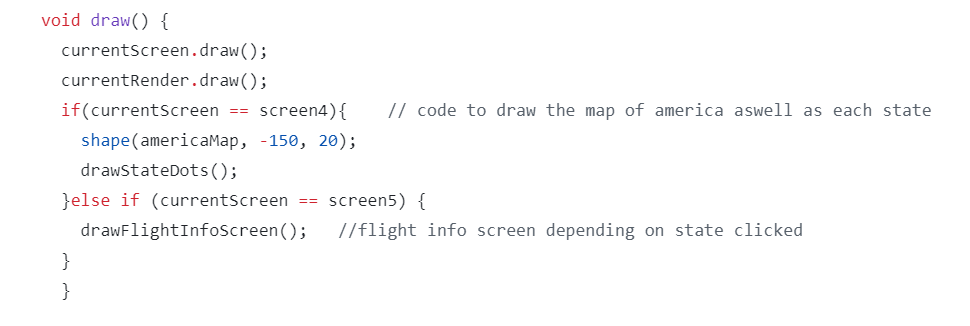
Main begins with a series of global variable declarations that will be used across the program. It then creates the window based on constants before entering void setup() where we take advantage of it only being invoked once to initialise many variables and read in and preprocess the data for our queries.



Ensuring that many of these relatively intensive operations only happen once, our program avoids lagging while running.

Main’s draw calls the currentScreen and currentRender to draw themselves.

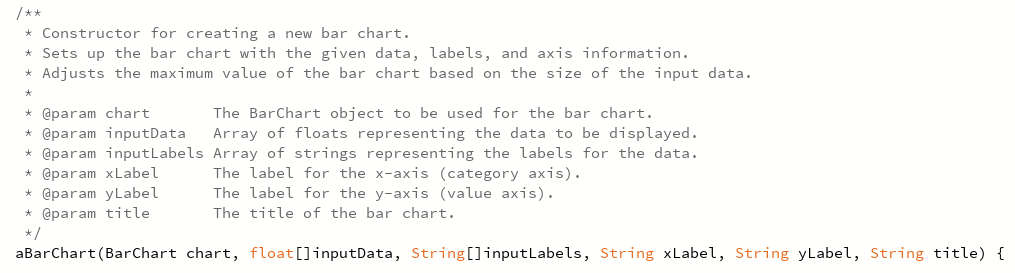
The draw() function first renders the current screen and its contents using currentScreen.draw() and currentRender.draw(). Then, it checks if screen4 is active to draw the US map and state points. If so, it calls drawStateDots().



The controlEvent(ControlEvent theEvent) function manages dropdown menu selections, displaying relevant data based on user choice.



**BarCharts**

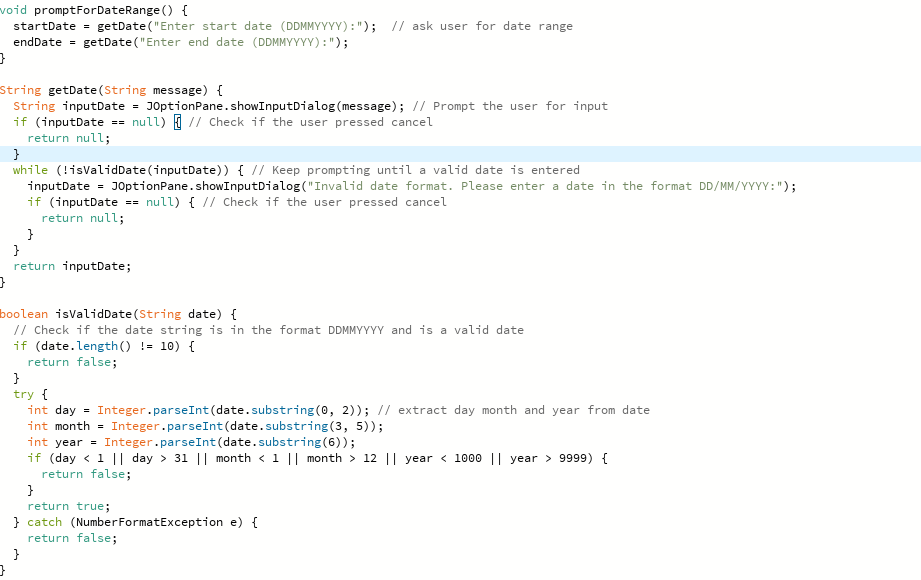
Implementing the gicentre.utils library, this class allows for the creation of modular bar chart objects which contain their own data, labels, and their own draw().

**Map**

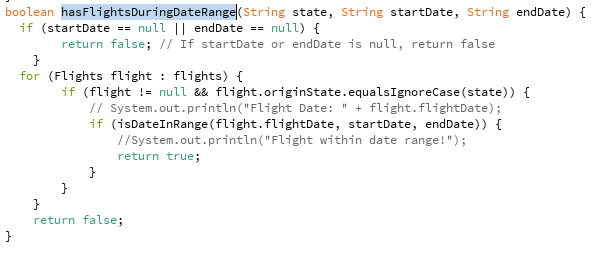
The Map section of our code consists of several functions aimed at loading the map, prompting the user for date range, validating the date input, checking for flights within the specified date range, drawing state dots on the map, handling user clicks on the map, and displaying flight information.

The loadMap loads the outline of the map of America from an SVG file and scales it to fit within the canvas. It also initialises a HashMap to store the coordinates of dots representing each state on the map.

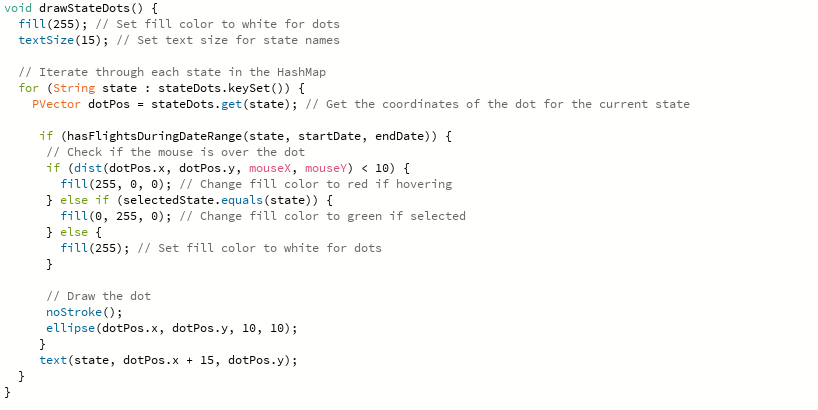
Then the promptForDateRange function asks the user to enter a start and end date in the format DD/MM/YYYY. It utilises the getDate() function to ensure valid date input from the user. Then it uses the isValidDate function to check whether the provided date string is in the correct format and represents a valid date.



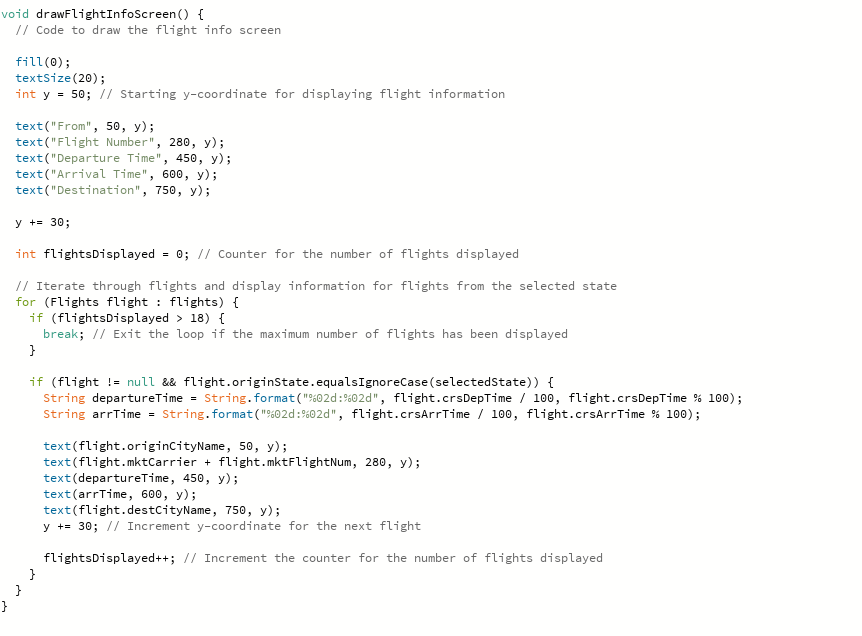
The hasFlightsDuringDateRange function extracts the relevant data from the flight object and loops through all flights and checks if they are within the flight range entered by the user.



Then the drawStateDots function is called and it draws dots representing each state on the map, highlighting them based on user interactions and the presence of flights within the specified date range.

****

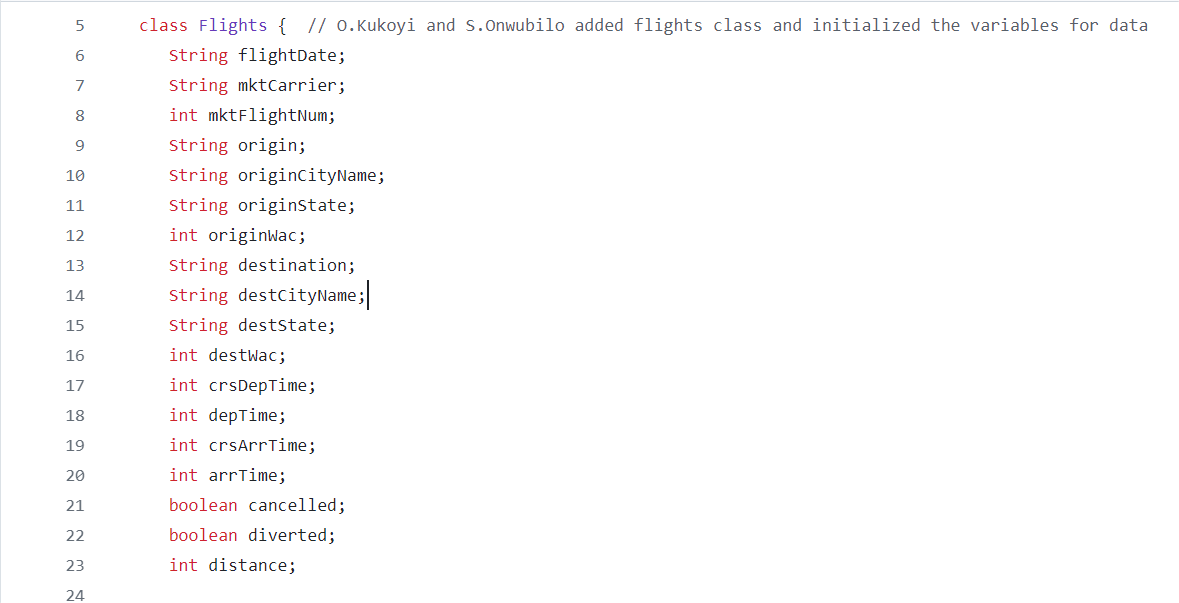
When any of these dots are clicked on, a flight information screen pops up, displaying details such as origin, flight number, departure, and arrival times for flights from the selected state.

****

**dataPoints**

The “Flights” class is intended to represent a single flight record in a flight tracking system. It encapsulates various properties related to a flight, such as the flight date ,carrier, flight number, origin and so on.

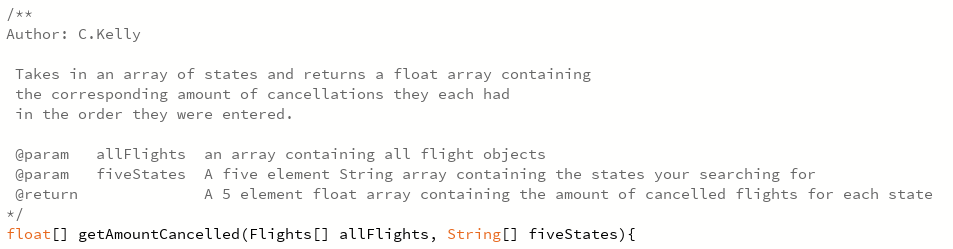
It includes:

****

The class provides a constructor that initialises these attributes with data from an input array. It facilitates storing and manipulating flight data within a flight tracking system.

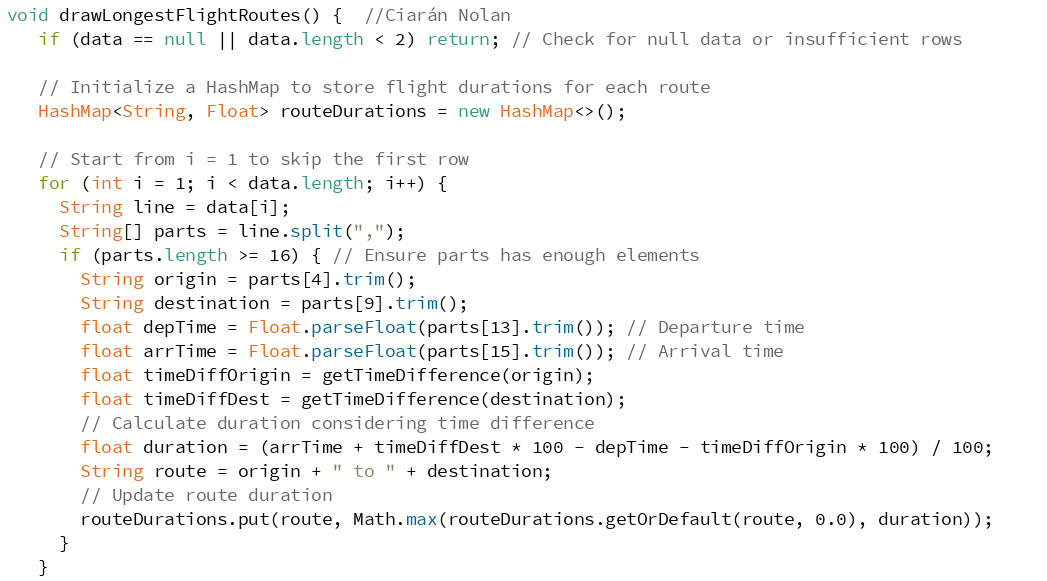
**dataProcessing**

Contains functions used to subdivide our data into smaller arrays for use in user queries and to be fed into barChart objects to create graphs.

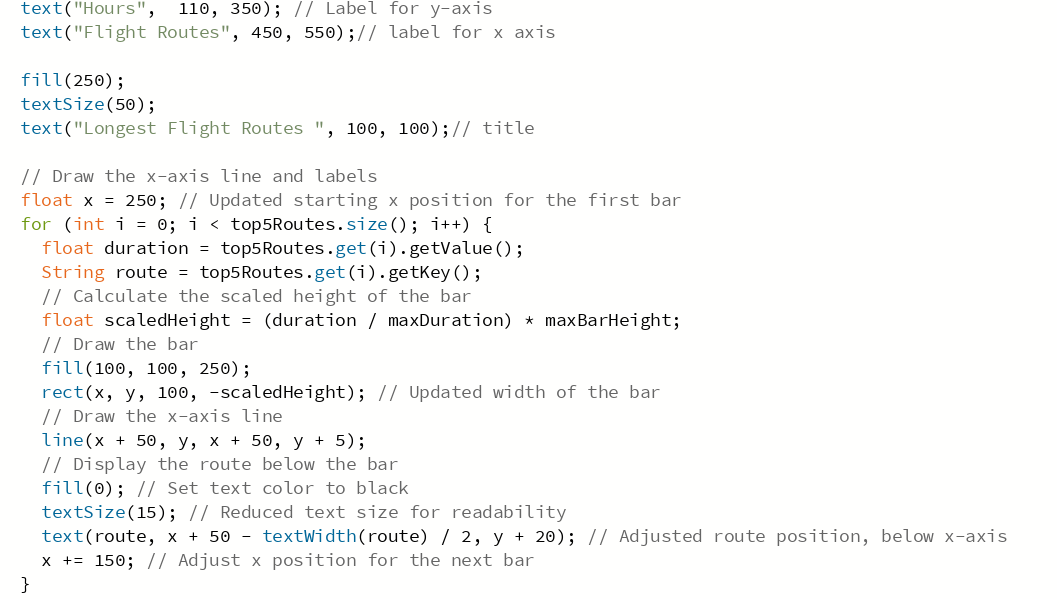


**Rendering**

The function drawLongestFlightRoutes() queries the data and calculates the longest flight routes. The for loop ensures that all the data is analysed except line 1 as it contains headings. Route Durations (HashMap) ensures that no 1 flight route occurs twice in the data.



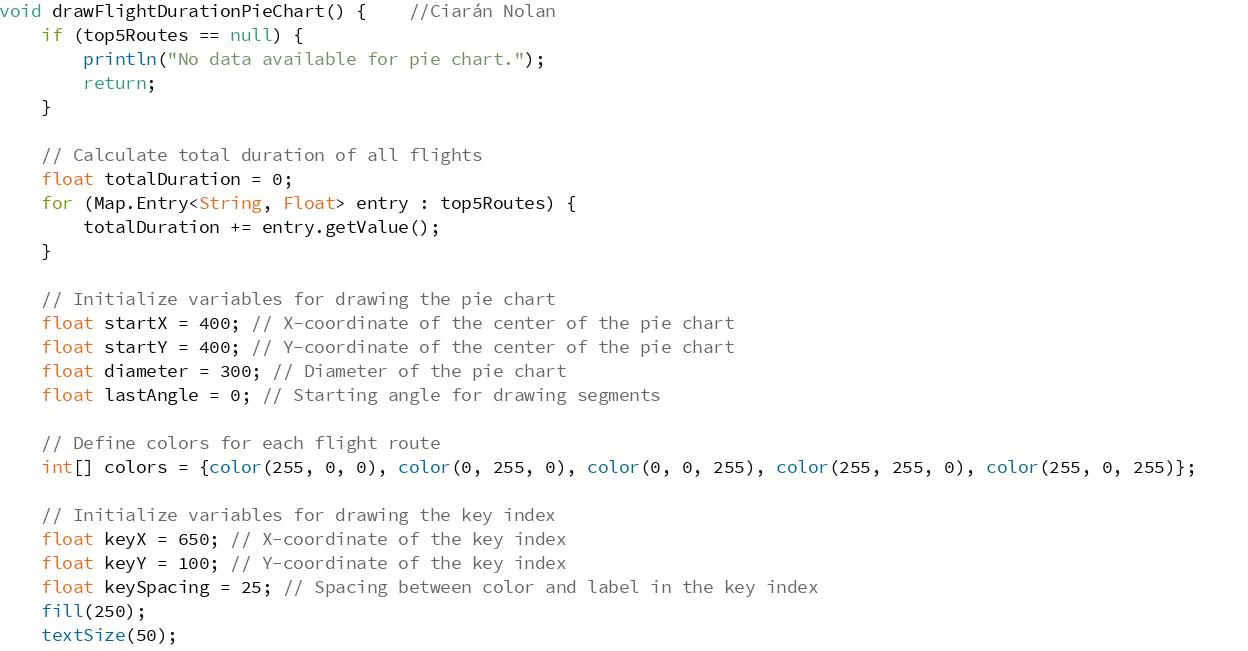
This draws a bar chart of the data.

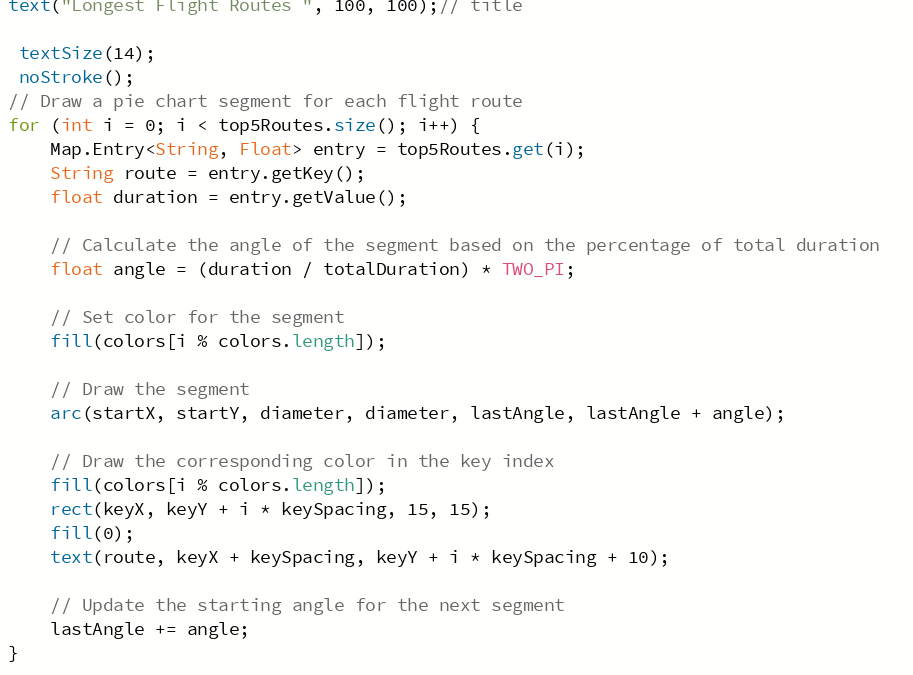


This accounts for the time difference between different locations.



This draws the pie chart of the flight route based on flight duration. It has its own method so that it can be called separately to drawLongestFlightRoutes().





The second half of the rendering class includes the function:

void drawBusiestAirports()

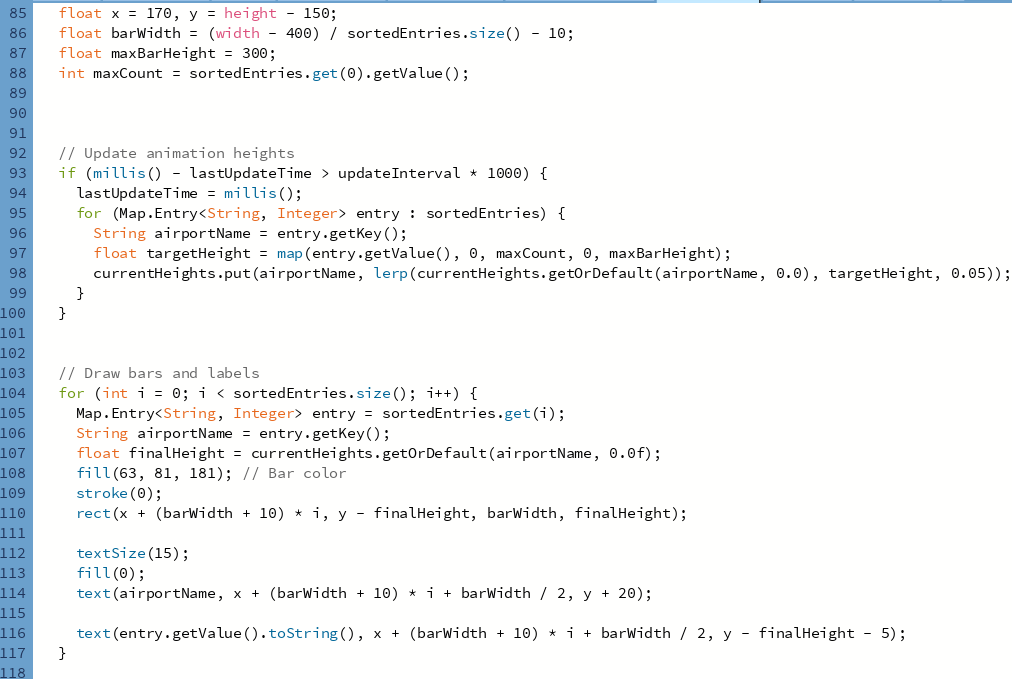
This function is responsible for the visual representation of the busiest airports. I decided to approach this by representing the data with a bar chart. The code is is broken down in this way:

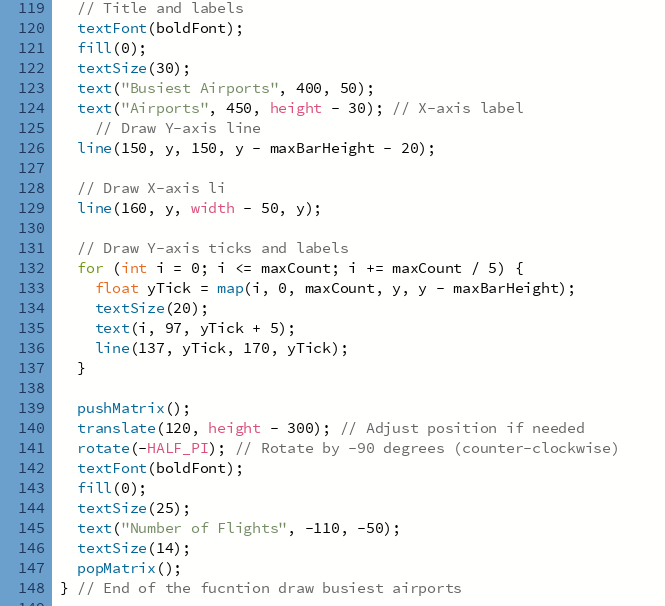
1. Reading and organising the data:



The preceding code reads in and sifts through the data. It gathers relevant data like the name of the departing and destination airport. Then, depending on the amount of time that specific name has appeared, it adds the total number to the airportCounts HashMap. The second half uses sortedEntries to limit the number of airports shown in the bar chart. This can be seen in the very last line of the code above, however this data can be adjusted to 5, 10, 20 or however many airports we wish to show.

1. The Animation

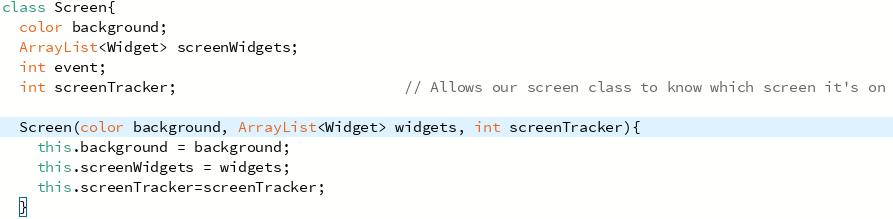




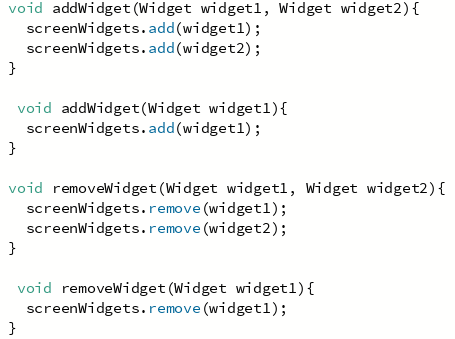
The code above describes how the animation was drawn and animated. In summary, it extracts how much a particular airport has been counted, and then draws how tall the bar chart is.

**Screens**

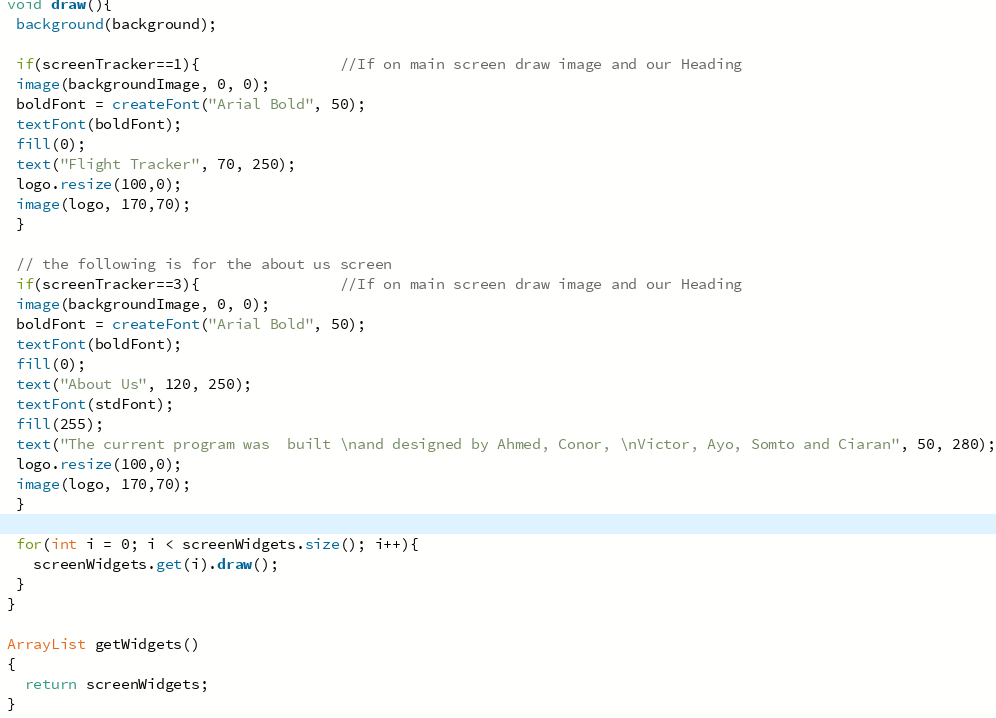
The ‘Screen’ class acts as the program’s interface, incorporating fonts, logos and the background image.



The class sets the background colour and manages a widget list using an ArrayList. The screenTracker variable allows the screen class to know which screen is being represented. Th constructor initialises the background, widgets and screenTracker variables.

****

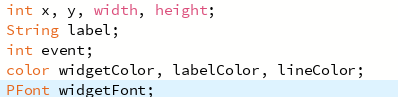
The widget handling methods manage the addition/removal of the widgets on the screen.



The draw() method draws the screen’s content based on the value of the ‘screenTracker’. If it’s 1 then the main menu is displayed but if it’s 3 then the ‘About us’ screen is shown. The method iterates through each widget in ‘screenWidgets’ and its ‘draw()’ method, drawing each widget. The ‘getWidgets()’ returns the list of widgets on the screen.

**Widget**

The class widget serves its purpose for creating the buttons for screen navigation.



It initialises the dimensions, colour and font through instance variables and the constructor.



The draw() method creates the widget as an entity on screen with the label included inside the rectangle. The getEvent() checks if the method is over the widget and returns the event associated or a null event. The last methods adjust the colour for when the mouse is over or not over the widget.