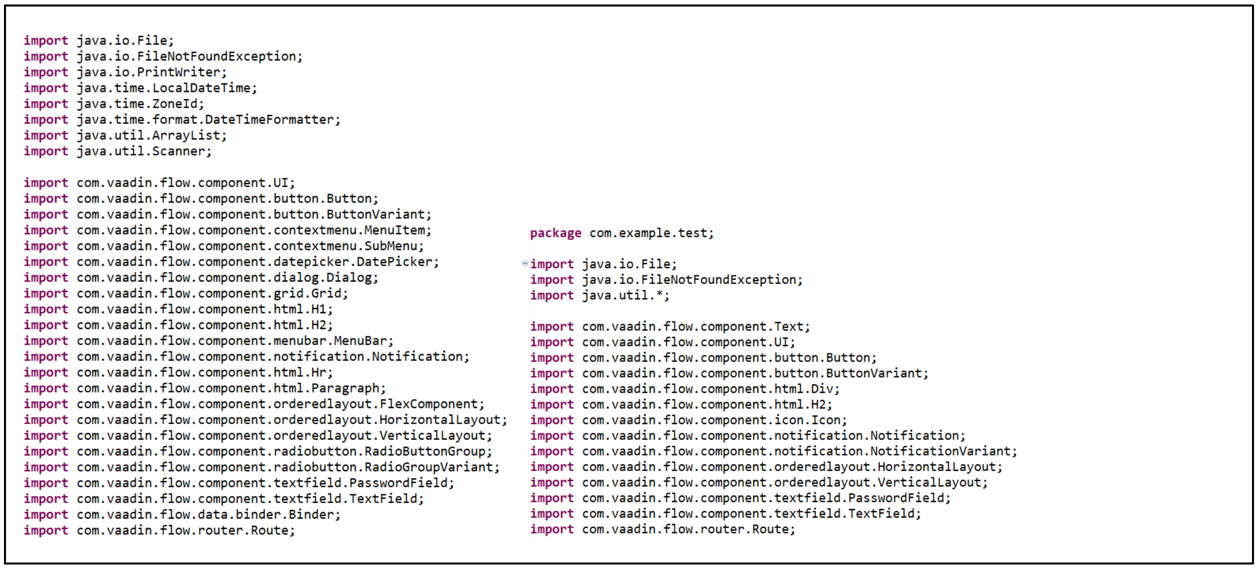
Criterion C: Development

The following points 1 to 8 display the techniques used in the program.

1: Use of third-party libraries - Vaadin

I used a third-party library, Vaadin[1], which helped in implementing my program and setting up a clear layout in which I could work. It allowed me to bring my ideas together in the form of a modern well-designed website that was easily navigable by my client. I made use of Vaadin by installing Eclipse[2], an IDE, and installing the Vaadin plug-in on it. I then imported many libraries (Figure 1) which allowed the use of components in my program such as buttons and dialogues (Figures 2, 3). This helped build the graphical user interface that my client could then use.



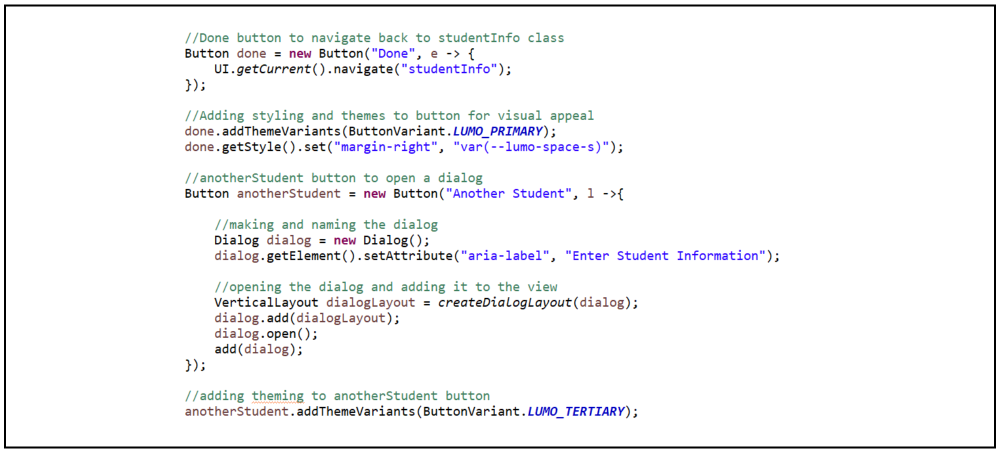
```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.Scanner;

import com.vaadin.flow.component.UI;
import com.vaadin.flow.component.button.Button;
import com.vaadin.flow.component.button.ButtonVariant;
import com.vaadin.flow.component.contextmenu.MenuItem;
import com.vaadin.flow.component.contextmenu.SubMenu;
import com.vaadin.flow.component.datepicker.DatePicker;
import com.vaadin.flow.component.dialog.Dialog;
import com.vaadin.flow.component.grid.Grid;
import com.vaadin.flow.component.html.H1;
import com.vaadin.flow.component.html.H2;
import com.vaadin.flow.component.menubar.MenuBar;
import com.vaadin.flow.component.notification.Notification;
import com.vaadin.flow.component.html.Hr;
import com.vaadin.flow.component.html.Paragraph;
import com.vaadin.flow.component.orderedlayout.FlexComponent;
import com.vaadin.flow.component.orderedlayout.HorizontalLayout;
import com.vaadin.flow.component.orderedlayout.VerticalLayout;
import com.vaadin.flow.component.radiobutton.RadioButtonGroup;
import com.vaadin.flow.component.radiobutton.RadioGroupVariant;
import com.vaadin.flow.component.textfield.PasswordField;
import com.vaadin.flow.component.textfield.TextField;
import com.vaadin.flow.data.binder.Binder;
import com.vaadin.flow.router.Route;
```

```
package com.example.test;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.*;

import com.vaadin.flow.component.Text;
import com.vaadin.flow.component.UI;
import com.vaadin.flow.component.button.Button;
import com.vaadin.flow.component.button.ButtonVariant;
import com.vaadin.flow.component.html.Div;
import com.vaadin.flow.component.html.H2;
import com.vaadin.flow.component.icon.Icon;
import com.vaadin.flow.component.notification.Notification;
import com.vaadin.flow.component.notification.NotificationVariant;
import com.vaadin.flow.component.orderedlayout.HorizontalLayout;
import com.vaadin.flow.component.orderedlayout.VerticalLayout;
import com.vaadin.flow.component.textfield.PasswordField;
import com.vaadin.flow.component.textfield.TextField;
import com.vaadin.flow.router.Route;
```

Figure 1. Importing Vaadin libraries in the Menu and Login class.

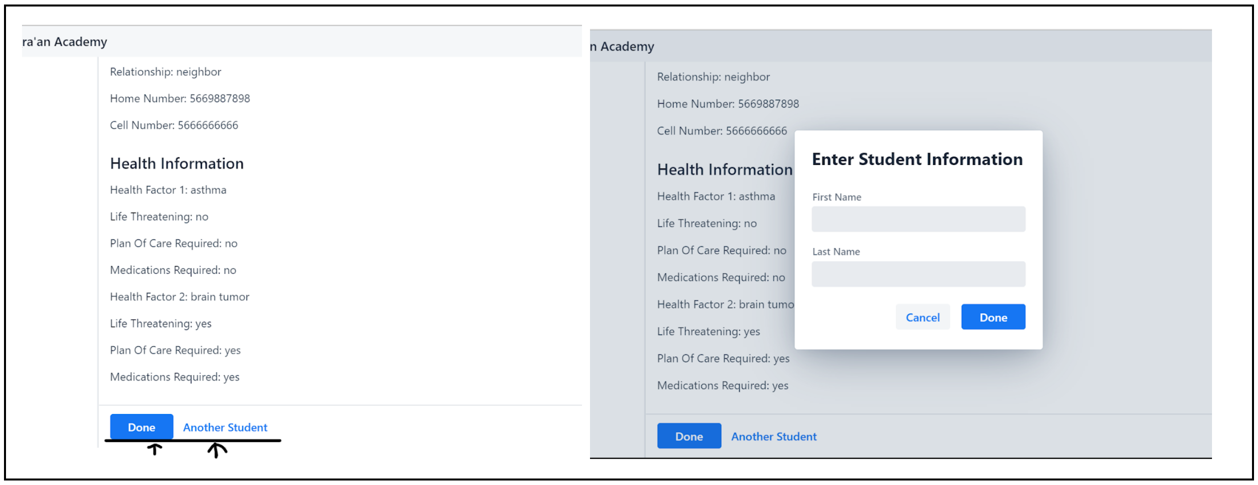
[1] Widenius, M. (n.d.). An open platform for building web apps in Java. Vaadin. Retrieved March 20, 2022, from https://vaadin.com/

[2] Eclipse Foundation, I. (n.d.). The community for Open Innovation and Collaboration: The eclipse foundation. Eclipse RSS. Retrieved March 18, 2022, from https://www.eclipse.org/



```
//Done button to navigate back to studentInfo class
Button done = new Button("Done", e -> {
    UI.getCurrent().navigate("studentInfo");
});

//Adding styling and themes to button for visual appeal
done.addThemeVariants(ButtonVariant.LUMO_PRIMARY);
done.getStyle().set("margin-right", "var(--lumo-space-s)");

//anotherStudent button to open a dialog
Button anotherStudent = new Button("Another Student", l ->{

    //making and naming the dialog
    Dialog dialog = new Dialog();
    dialog.getElement().setAttribute("aria-label", "Enter Student Information");

    //opening the dialog and adding it to the view
    VerticalLayout dialogLayout = createDialogLayout(dialog);
    dialog.add(dialogLayout);
    dialog.open();
    add(dialog);
});

//adding theming to anotherStudent button
anotherStudent.addThemeVariants(ButtonVariant.LUMO_TERTIARY);
```

Figure 2. The building of buttons and dialogue provided by the Vaadin libraries



ra'an Academy

Relationship: neighbor

Home Number: 5669887898

Cell Number: 5666666666

Health Information

Health Factor 1: asthma

Life Threatening: no

Plan Of Care Required: no

Medications Required: no

Health Factor 2: brain tumor

Life Threatening: yes

Plan Of Care Required: yes

Medications Required: yes

Done Another Student

n Academy

Relationship: neighbor

Home Number: 5669887898

Cell Number: 5666666666

Health Information

Health Factor 1: asthma

Life Threatening: no

Plan Of Care Required: no

Medications Required: no

Health Factor 2: brain tumo

Life Threatening: yes

Plan Of Care Required: yes

Medications Required: yes

Enter Student Information

First Name

Last Name

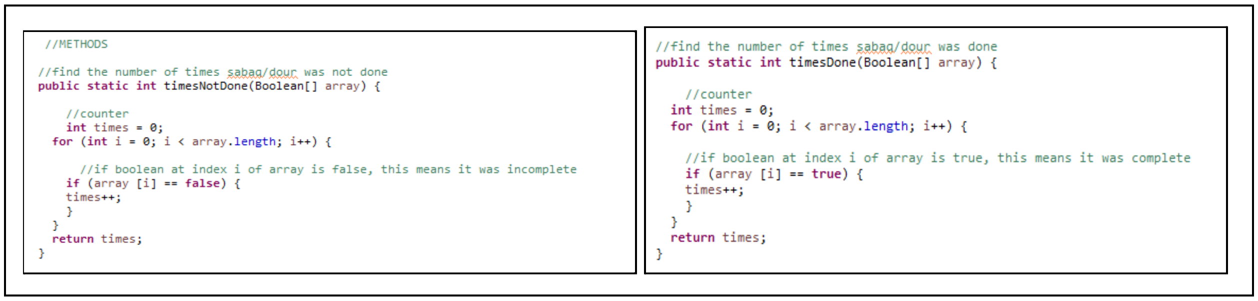
Cancel Done

Done Another Student

Figure 3. Buttons and dialogue displayed in the user interface

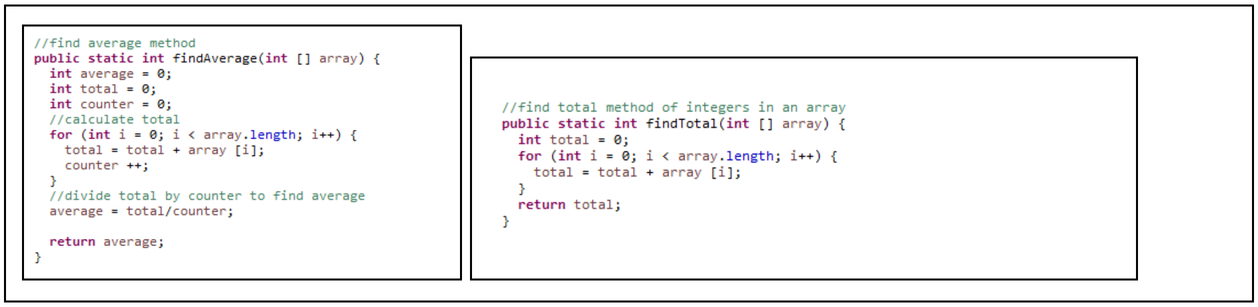
2: Abstract data structures - Arrays, ArrayLists

I used two abstract data structures in my program, arrays and Arraylists, which were vital in the functioning of the program. Arrays were used to calculate and display monthly progress, one of the main needs of the client (Figure 4, 5), while Arraylists held student objects and records of attendance and COVID screenings (Figure 6). Both these data structures organized the program and allowed it to function.



```
 //METHODS

//find the number of times sabaq/dour was not done
public static int timesNotDone(Boolean[] array) {

    //counter
    int times = 0;
  for (int i = 0; i < array.length; i++) {

      //if boolean at index i of array is false, this means it was incomplete
    if (array [i] == false) {
    times++;
    }
  }
  return times;
}
```

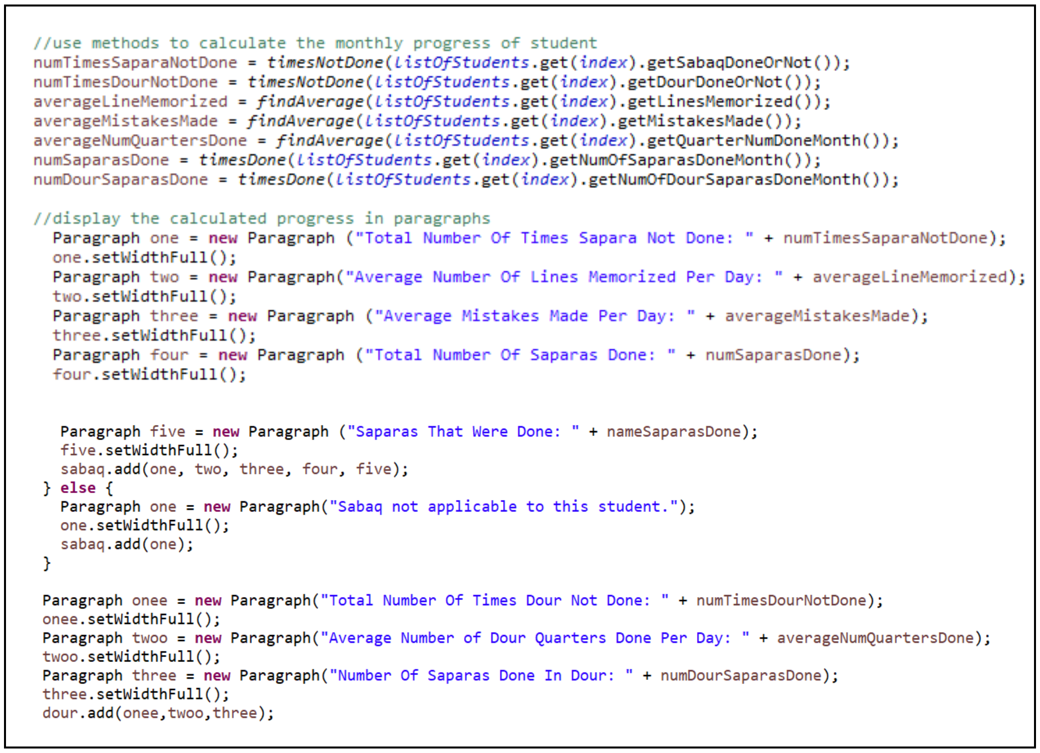
```
//find the number of times sabaq/dour was done
public static int timesDone(Boolean[] array) {

    //counter
  int times = 0;
  for (int i = 0; i < array.length; i++) {

    //if boolean at index i of array is true, this means it was complete
    if (array [i] == true) {
    times++;
    }
  }
  return times;
}
```



```java
//find average method
public static int findAverage(int [] array) {
  int average = 0;
  int total = 0;
  int counter = 0;
  //calculate total
  for (int i = 0; i < array.length; i++) {
    total = total + array [i];
    counter ++;
  }
  //divide total by counter to find average
  average = total/counter;

  return average;
}
```

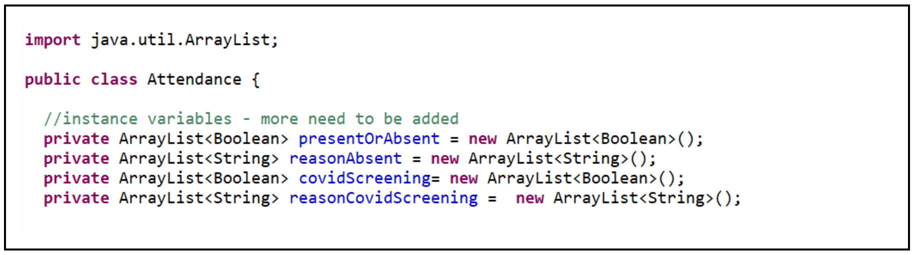
```java
//find total method of integers in an array
public static int findTotal(int [] array) {
  int total = 0;
  for (int i = 0; i < array.length; i++) {
    total = total + array [i];
  }
  return total;
}
```

Figure 4. Calculation of monthly progress methods through arrays



```java
//use methods to calculate the monthly progress of student
numTimesSaparaNotDone = timesNotDone(listOfStudents.get(index).getSabaqDoneOrNot());
numTimesDourNotDone = timesNotDone(listOfStudents.get(index).getDourDoneOrNot());
averageLineMemorized = findAverage(listOfStudents.get(index).getLinesMemorized());
averageMistakesMade = findAverage(listOfStudents.get(index).getMistakesMade());
averageNumQuartersDone = findAverage(listOfStudents.get(index).getQuarterNumDoneMonth());
numSaparasDone = timesDone(listOfStudents.get(index).getNumOfSaparasDoneMonth());
numDourSaparasDone = timesDone(listOfStudents.get(index).getNumOfDourSaparasDoneMonth());

//display the calculated progress in paragraphs
  Paragraph one = new Paragraph ("Total Number Of Times Sapara Not Done: " + numTimesSaparaNotDone);
  one.setWidthFull();
  Paragraph two = new Paragraph("Average Number Of Lines Memorized Per Day: " + averageLineMemorized);
  two.setWidthFull();
  Paragraph three = new Paragraph ("Average Mistakes Made Per Day: " + averageMistakesMade);
  three.setWidthFull();
  Paragraph four = new Paragraph ("Total Number Of Saparas Done: " + numSaparasDone);
  four.setWidthFull();


   Paragraph five = new Paragraph ("Saparas That Were Done: " + nameSaparasDone);
   five.setWidthFull();
   sabaq.add(one, two, three, four, five);
 } else {
   Paragraph one = new Paragraph("Sabaq not applicable to this student.");
   one.setWidthFull();
   sabaq.add(one);
 }

 Paragraph onee = new Paragraph("Total Number Of Times Dour Not Done: " + numTimesDourNotDone);
 onee.setWidthFull();
 Paragraph twoo = new Paragraph("Average Number of Dour Quarters Done Per Day: " + averageNumQuartersDone);
 twoo.setWidthFull();
 Paragraph three = new Paragraph("Number Of Saparas Done In Dour: " + numDourSaparasDone);
 three.setWidthFull();
 dour.add(onee,twoo,three);
```

Figure 5. Implementation of calculation methods

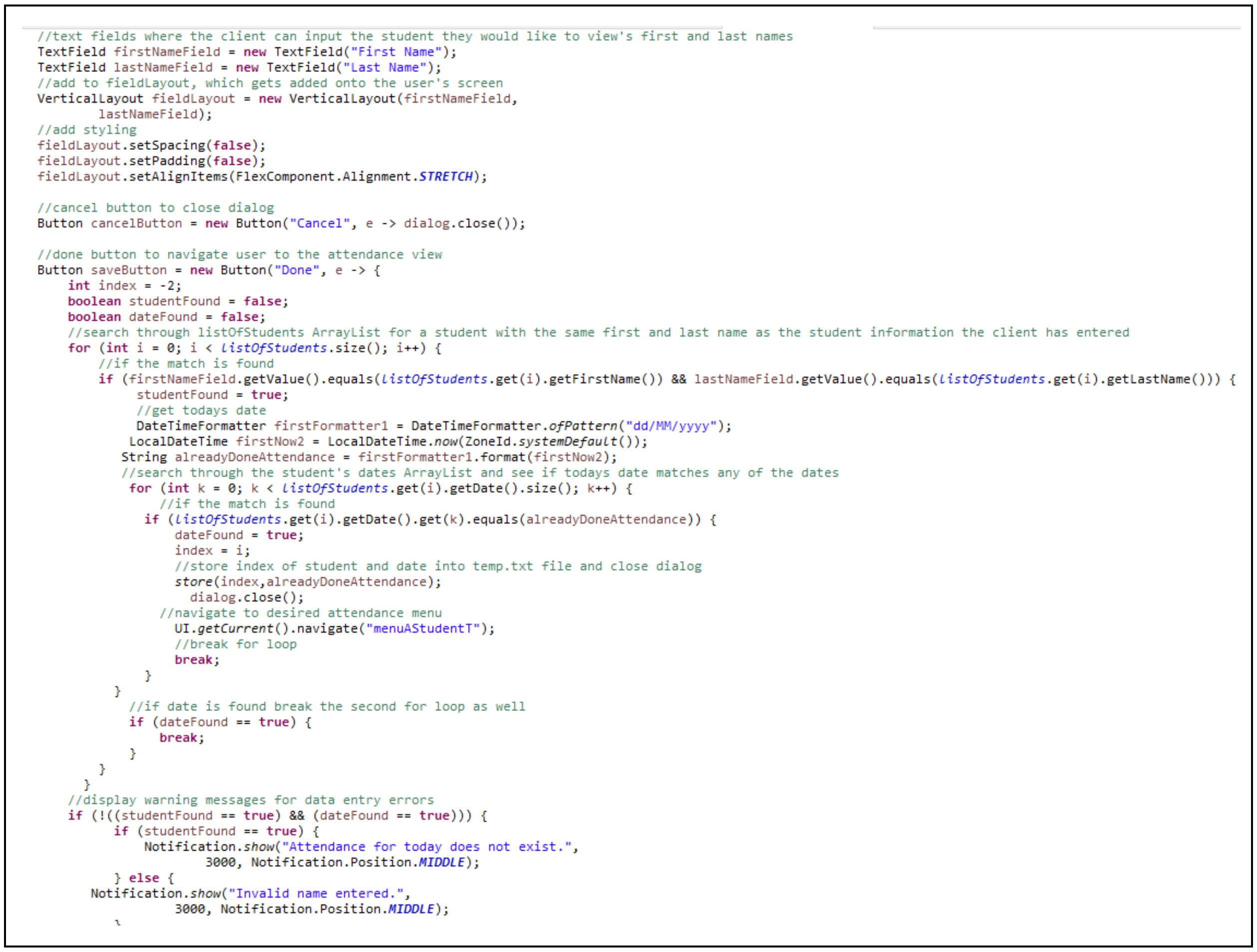


```java
import java.util.ArrayList;

public class Attendance {

  //instance variables - more need to be added
  private ArrayList<Boolean> presentOrAbsent = new ArrayList<Boolean>();
  private ArrayList<String> reasonAbsent = new ArrayList<String>();
  private ArrayList<Boolean> covidScreening= new ArrayList<Boolean>();
  private ArrayList<String> reasonCovidScreening =  new ArrayList<String>();
```

Figure 6. The use of ArrayLists in attendance class

3: Algorithms - Nested loops and if statements

One of the requirements of this program, as stated by the client (Appendix III), was to display a particular student's attendance of the current day. I used nested loops and if statements to make this algorithm and it works by taking in student information and searching for a particular student in an ArrayList of students. Then, it searches for the current day in the student's dates array (Figure 7), if it is found, the client is navigated to a page with the attendance displayed, otherwise it outputs that attendance is incomplete. If any entry error occurs, a warning message is displayed. This algorithm was crucial to the well-development of the program as it gave the client the option to change a student's attendance if needed, for example, if a student attended the class late after the attendance is submitted.

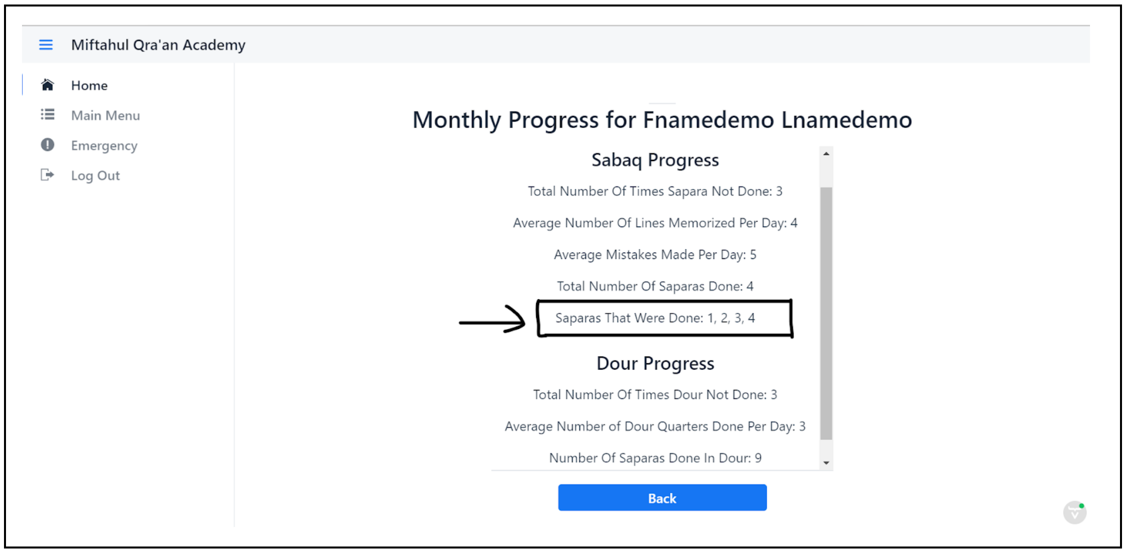


```java
//text fields where the client can input the student they would like to view's first and last names
TextField firstNameField = new TextField("First Name");
TextField lastNameField = new TextField("Last Name");
//add to fieldLayout, which gets added onto the user's screen
VerticalLayout fieldLayout = new VerticalLayout(firstNameField,
        lastNameField);
//add styling
fieldLayout.setSpacing(false);
fieldLayout.setPadding(false);
fieldLayout.setAlignItems(FlexComponent.Alignment.STRETCH);

//cancel button to close dialog
Button cancelButton = new Button("Cancel", e -> dialog.close());

//done button to navigate user to the attendance view
Button saveButton = new Button("Done", e -> {
    int index = -2;
    boolean studentFound = false;
    boolean dateFound = false;
    //search through listOfStudents ArrayList for a student with the same first and last name as the student information the client has entered
    for (int i = 0; i < listOfStudents.size(); i++) {
        //if the match is found
        if (firstNameField.getValue().equals(listOfStudents.get(i).getFirstName()) && lastNameField.getValue().equals(listOfStudents.get(i).getLastName())) {
             studentFound = true;
             //get todays date
             DateTimeFormatter firstFormatter1 = DateTimeFormatter.ofPattern("dd/MM/yyyy");
            LocalDateTime firstNow2 = LocalDateTime.now(ZoneId.systemDefault());
           String alreadyDoneAttendance = firstFormatter1.format(firstNow2);
           //search through the student's dates ArrayList and see if todays date matches any of the dates
            for (int k = 0; k < listOfStudents.get(i).getDate().size(); k++) {
                //if the match is found
              if (listOfStudents.get(i).getDate().get(k).equals(alreadyDoneAttendance)) {
                  dateFound = true;
                  index = i;
                  //store index of student and date into temp.txt file and close dialog
                  store(index,alreadyDoneAttendance);
                    dialog.close();
                //navigate to desired attendance menu
                  UI.getCurrent().navigate("menuAStudentT");
                  //break for loop
                  break;
              }
          }
            //if date is found break the second for loop as well
            if (dateFound == true) {
                break;
            }
        }
    }
    //display warning messages for data entry errors
    if (!((studentFound == true) && (dateFound == true))) {
          if (studentFound == true) {
              Notification.show("Attendance for today does not exist.",
                      3000, Notification.Position.MIDDLE);
          } else {
       Notification.show("Invalid name entered.",
                  3000, Notification.Position.MIDDLE);
          }
```

Figure 7. the displaying of a student's attendance of the current day algorithm

Technique 4: Sorting algorithms - Insertion and sequential sorts

I used two sorting algorithms, insertion and sequential sort, to display a more comprehensive list for the client of the Sapara numbers that the student has finished (Figure 8). I implemented insertion sort by setting the first index as sorted, then shifting each subsequent value into the correct position by finding the lowest value on the sorted side of the array at indices lower than the current index (Figure 9). I implemented sequential sort by repeatedly finding the minimum value, and moving it to the front of the array (Figure 10). Both these instances sorted the array in ascending order, which was fit for the client's needs from the program.



Monthly Progress for Fnamedemo Lnamedemo

Sabaq Progress

Total Number Of Times Sapara Not Done: 3

Average Number Of Lines Memorized Per Day: 4

Average Mistakes Made Per Day: 5

Total Number Of Saparas Done: 4

Saparas That Were Done: 1, 2, 3, 4

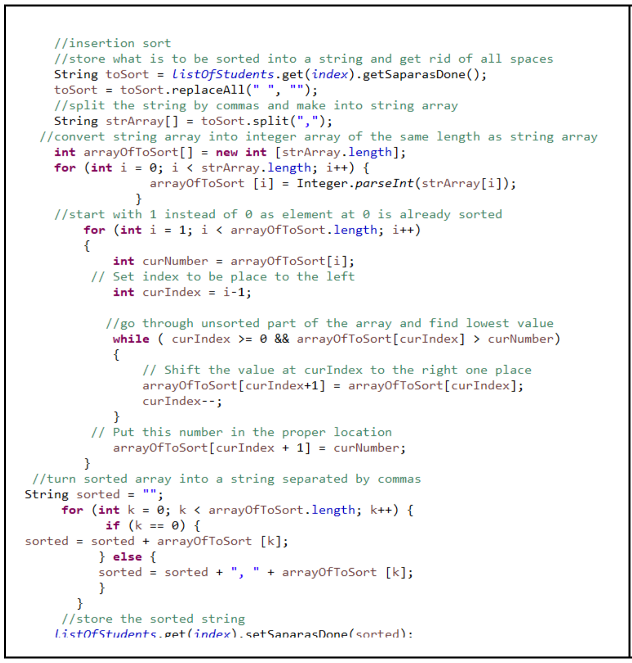
Dour Progress

Total Number Of Times Dour Not Done: 3

Average Number of Dour Quarters Done Per Day: 3

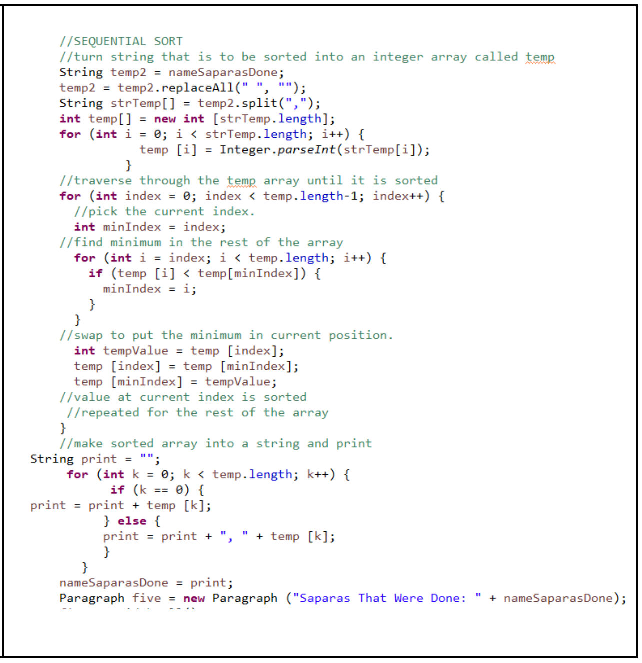
Number Of Saparas Done In Dour: 9

Figure 8. Sorted "Saparas That Were Done" in ascending order



```
  //insertion sort
  //store what is to be sorted into a string and get rid of all spaces
  String toSort = listOfStudents.get(index).getSaparasDone();
  toSort = toSort.replaceAll(" ", "");
  //split the string by commas and make into string array
  String strArray[] = toSort.split(",");
//convert string array into integer array of the same length as string array
  int arrayOfToSort[] = new int [strArray.length];
  for (int i = 0; i < strArray.length; i++) {
             arrayOfToSort [i] = Integer.parseInt(strArray[i]);
           }
  //start with 1 instead of 0 as element at 0 is already sorted
      for (int i = 1; i < arrayOfToSort.length; i++)
      {
          int curNumber = arrayOfToSort[i];
       // Set index to be place to the left
          int curIndex = i-1;

         //go through unsorted part of the array and find lowest value
          while ( curIndex >= 0 && arrayOfToSort[curIndex] > curNumber)
          {
              // Shift the value at curIndex to the right one place
              arrayOfToSort[curIndex+1] = arrayOfToSort[curIndex];
              curIndex--;
          }
       // Put this number in the proper location
          arrayOfToSort[curIndex + 1] = curNumber;
      }
 //turn sorted array into a string separated by commas
String sorted = "";
     for (int k = 0; k < arrayOfToSort.length; k++) {
         if (k == 0) {
sorted = sorted + arrayOfToSort [k];
         } else {
         sorted = sorted + ", " + arrayOfToSort [k];
         }
     }
   //store the sorted string
  listOfStudents.get(index).setSaparasDone(sorted);
```

Figure 9. Implementation of insertion sort

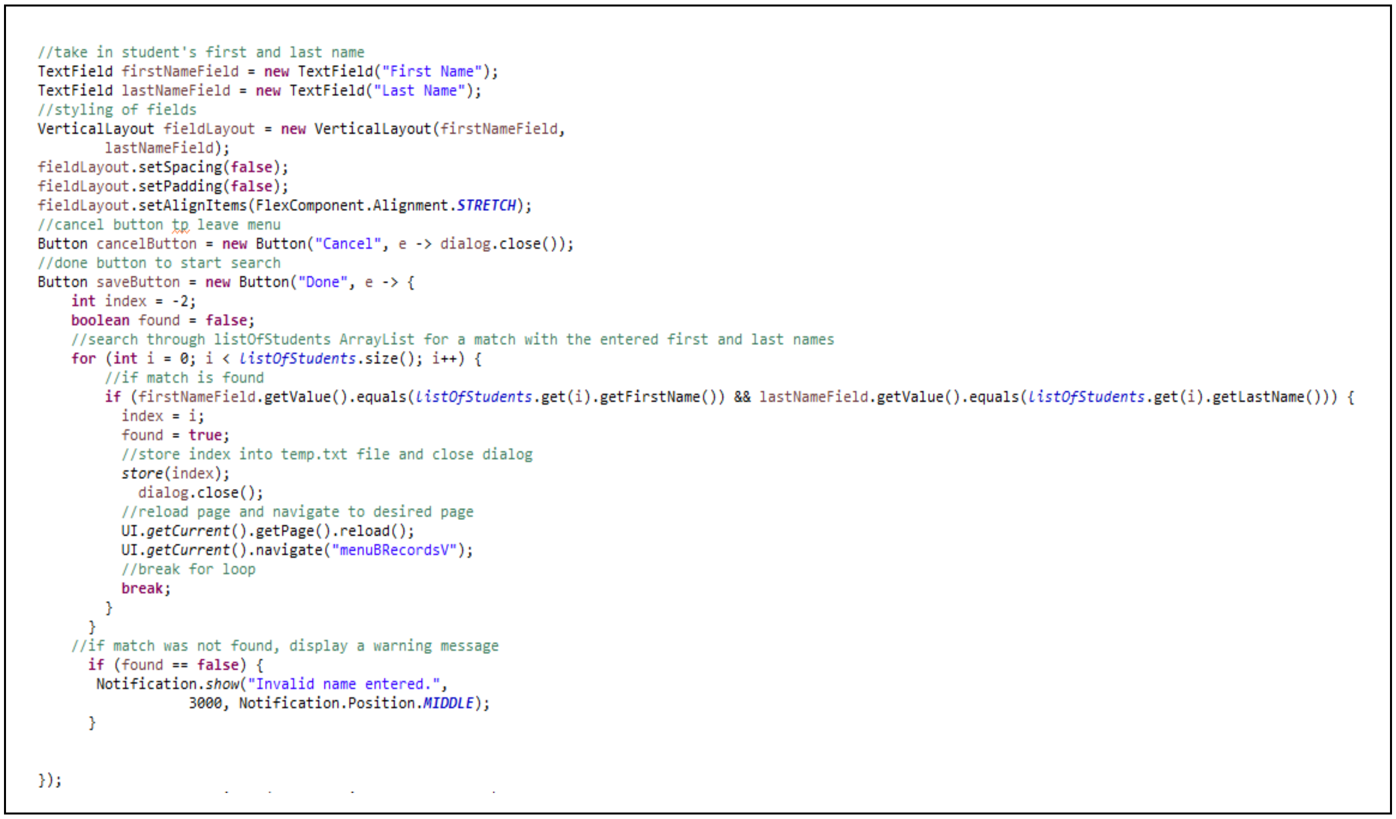


```
  //SEQUENTIAL SORT
  //turn string that is to be sorted into an integer array called temp
  String temp2 = nameSaparasDone;
  temp2 = temp2.replaceAll(" ", "");
  String strTemp[] = temp2.split(",");
  int temp[] = new int [strTemp.length];
  for (int i = 0; i < strTemp.length; i++) {
             temp [i] = Integer.parseInt(strTemp[i]);
           }
  //traverse through the temp array until it is sorted
  for (int index = 0; index < temp.length-1; index++) {
    //pick the current index.
    int minIndex = index;
  //find minimum in the rest of the array
    for (int i = index; i < temp.length; i++) {
      if (temp [i] < temp[minIndex]) {
        minIndex = i;
      }
    }
  //swap to put the minimum in current position.
    int tempValue = temp [index];
    temp [index] = temp [minIndex];
    temp [minIndex] = tempValue;
  //value at current index is sorted
   //repeated for the rest of the array
  }
  //make sorted array into a string and print
String print = "";
     for (int k = 0; k < temp.length; k++) {
          if (k == 0) {
print = print + temp [k];
         } else {
         print = print + ", " + temp [k];
         }
      }
  nameSaparasDone = print;
  Paragraph five = new Paragraph ("Saparas That Were Done: " + nameSaparasDone);
```

Figure 10. Implementation of sequential sort

5: Searching algorithms - Linear search

I used a searching algorithm, linear search, for the searching and retrieval of a student of the client's choice from an ArrayList of student objects. I implemented this search by taking input of the student's first and last names, then checking each element in order until there was a match or the end of the array was reached, in which case I asked for the input again (Figure 11). This helped in the formation of a well-designed program as it allowed successful interaction between the client and the program. It also aided the program in the execution of its intended functions.



```
//take in student's first and last name
TextField firstNameField = new TextField("First Name");
TextField lastNameField = new TextField("Last Name");
//styling of fields
VerticalLayout fieldLayout = new VerticalLayout(firstNameField,
        lastNameField);
fieldLayout.setSpacing(false);
fieldLayout.setPadding(false);
fieldLayout.setAlignItems(FlexComponent.Alignment.STRETCH);
//cancel button tp leave menu
Button cancelButton = new Button("Cancel", e -> dialog.close());
//done button to start search
Button saveButton = new Button("Done", e -> {
    int index = -2;
    boolean found = false;
    //search through listOfStudents ArrayList for a match with the entered first and last names
    for (int i = 0; i < listOfStudents.size(); i++) {
        //if match is found
        if (firstNameField.getValue().equals(listOfStudents.get(i).getFirstName()) && lastNameField.getValue().equals(listOfStudents.get(i).getLastName())) {
          index = i;
          found = true;
          //store index into temp.txt file and close dialog
          store(index);
            dialog.close();
          //reload page and navigate to desired page
          UI.getCurrent().getPage().reload();
          UI.getCurrent().navigate("menuBRecordsV");
          //break for loop
          break;
        }
      }
    //if match was not found, display a warning message
      if (found == false) {
       Notification.show("Invalid name entered.",
                  3000, Notification.Position.MIDDLE);
      }


});
```

Figure 11. Linear search algorithm

6: Advanced Java techniques - Lambda functions

I used lambda functions as it uses an expression to represent a method interface in a clear and concise manner. It also helped me in keeping up with my thought process as they made the code more readable and kept it concise and compact. I used lambda functions many times throughout the code, like in implementing buttons (Figure 12, 13, 14) and for my grid display (Figure 15).

```
//emergency button utilizing lambda functions
Button emergencyButton = new Button("Emergency", e -> {
    //if clicked, make a new dialog
    Dialog dialog = new Dialog();
    dialog.getElement().setAttribute("aria-label", "Enter Student Information");

    VerticalLayout dialogLayout = createDialogLayout(dialog);
    dialog.add(dialogLayout);
    dialog.open();
    add(dialog);
});
```

Figure 12. Emergency button made using lambda functions

```
//start button utilizing lambda functions
Button startButton = new Button("Let's Get Started!", e-> {
    //if clicked, navigate to started class
    UI.getCurrent().navigate("started");
});
```

Figure 13. Start button using lambda functions

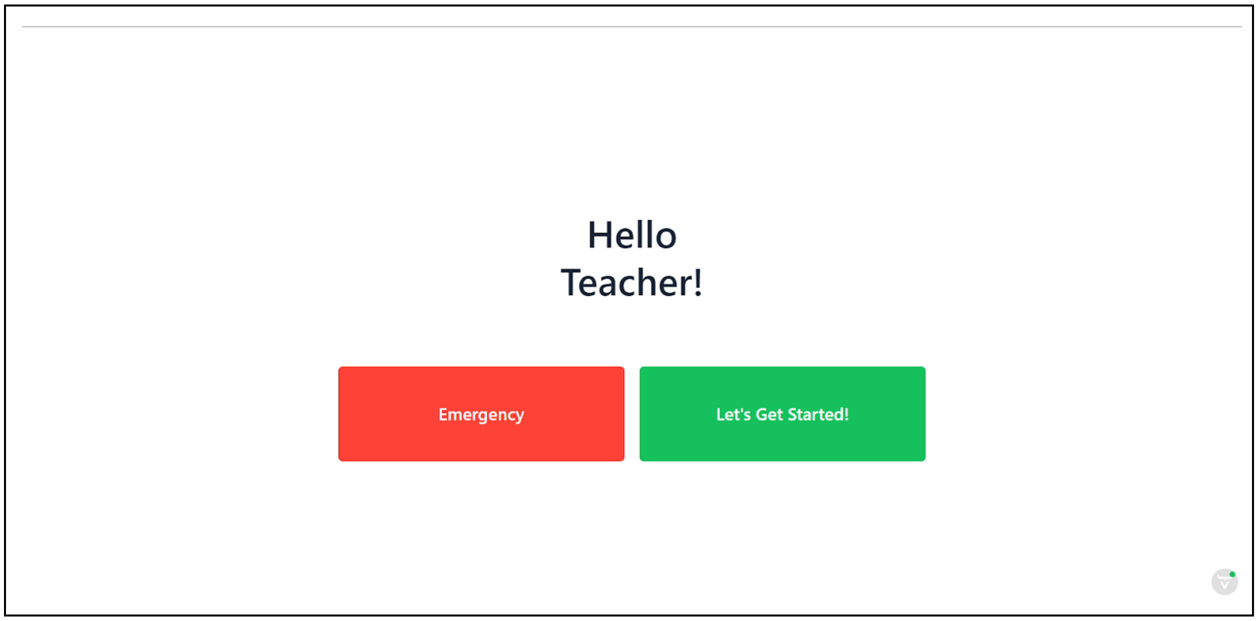
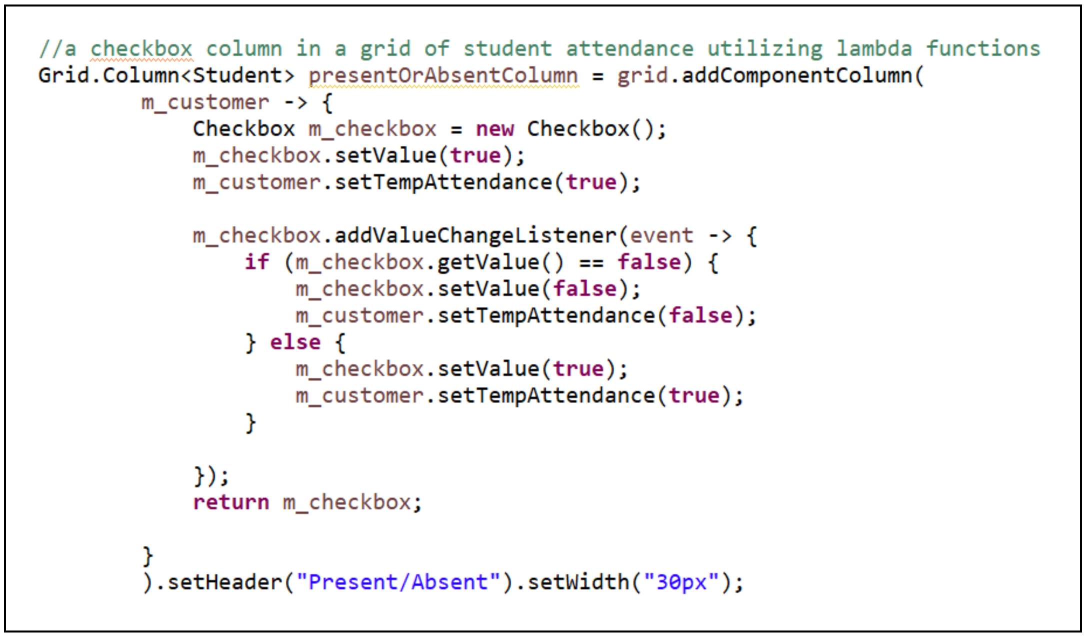


Figure 14. Both buttons on the user interface

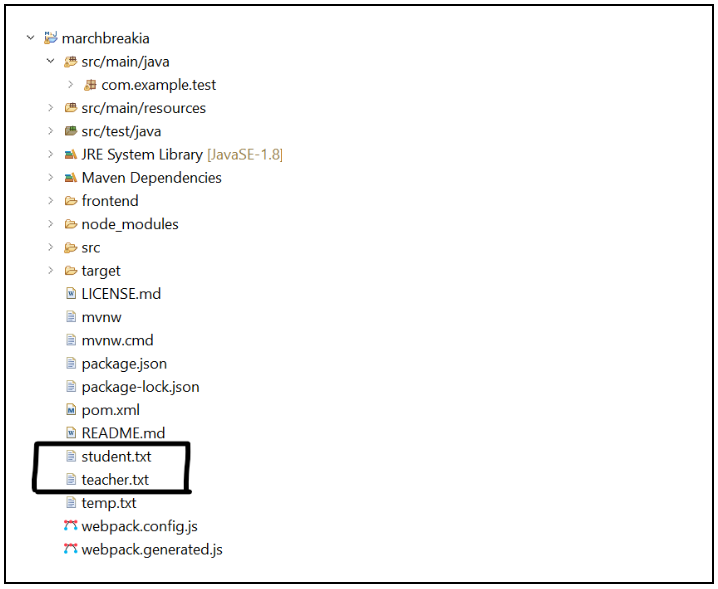


```java
//a checkbox column in a grid of student attendance utilizing lambda functions
Grid.Column<Student> presentOrAbsentColumn = grid.addComponentColumn(
        m_customer -> {
            Checkbox m_checkbox = new Checkbox();
            m_checkbox.setValue(true);
            m_customer.setTempAttendance(true);

            m_checkbox.addValueChangeListener(event -> {
                if (m_checkbox.getValue() == false) {
                    m_checkbox.setValue(false);
                    m_customer.setTempAttendance(false);
                } else {
                    m_checkbox.setValue(true);
                    m_customer.setTempAttendance(true);
                }

            });
            return m_checkbox;

        }
        ).setHeader("Present/Absent").setWidth("30px");
```

Figure 15. A column on a grid using lambda functions

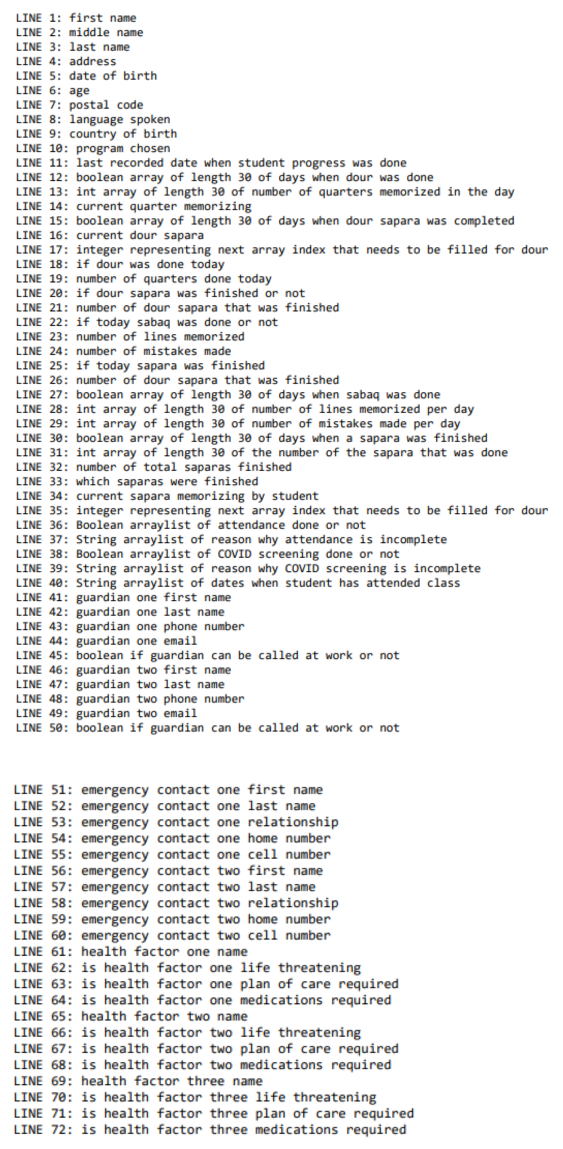
7: File Input and Output

I used file input and output to store all students' and the client's information in two text files (Figure 16, 17, 18). This allowed the client to open, change and save all information for later use. I implemented file input and output through the use of the libraries of scanners and print writers (Figure 19). File input works by storing all information of students and clients line by line into the appropriate text files while file output works by reading in the information line by line and making the teacher and student objects (Figure 20).



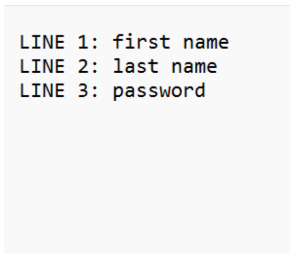
```
marchbreakia
  src/main/java
    com.example.test
  src/main/resources
  src/test/java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  frontend
  node_modules
  src
  target
  LICENSE.md
  mvnw
  mvnw.cmd
  package.json
  package-lock.json
  pom.xml
  README.md
  student.txt
  teacher.txt
  temp.txt
  webpack.config.js
  webpack.generated.js
```

Figure 16. The two text files used in file input and output



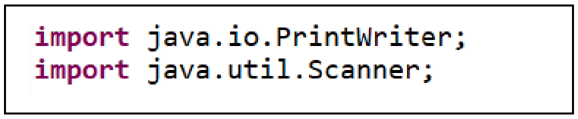
```
LINE 1: first name
LINE 2: middle name
LINE 3: last name
LINE 4: address
LINE 5: date of birth
LINE 6: age
LINE 7: postal code
LINE 8: language spoken
LINE 9: country of birth
LINE 10: program chosen
LINE 11: last recorded date when student progress was done
LINE 12: boolean array of length 30 of days when dour was done
LINE 13: int array of length 30 of number of quarters memorized in the day
LINE 14: current quarter memorizing
LINE 15: boolean array of length 30 of days when dour sapara was completed
LINE 16: current dour sapara
LINE 17: integer representing next array index that needs to be filled for dour
LINE 18: if dour was done today
LINE 19: number of quarters done today
LINE 20: if dour sapara was finished or not
LINE 21: number of dour sapara that was finished
LINE 22: if today sabaq was done or not
LINE 23: number of lines memorized
LINE 24: number of mistakes made
LINE 25: if today sapara was finished
LINE 26: number of dour sapara that was finished
LINE 27: boolean array of length 30 of days when sabaq was done
LINE 28: int array of length 30 of number of lines memorized per day
LINE 29: int array of length 30 of number of mistakes made per day
LINE 30: boolean array of length 30 of days when a sapara was finished
LINE 31: int array of length 30 of the number of the sapara that was done
LINE 32: number of total saparas finished
LINE 33: which saparas were finished
LINE 34: current sapara memorizing by student
LINE 35: integer representing next array index that needs to be filled for dour
LINE 36: Boolean arraylist of attendance done or not
LINE 37: String arraylist of reason why attendance is incomplete
LINE 38: Boolean arraylist of COVID screening done or not
LINE 39: String arraylist of reason why COVID screening is incomplete
LINE 40: String arraylist of dates when student has attended class
LINE 41: guardian one first name
LINE 42: guardian one last name
LINE 43: guardian one phone number
LINE 44: guardian one email
LINE 45: boolean if guardian can be called at work or not
LINE 46: guardian two first name
LINE 47: guardian two last name
LINE 48: guardian two phone number
LINE 49: guardian two email
LINE 50: boolean if guardian can be called at work or not


LINE 51: emergency contact one first name
LINE 52: emergency contact one last name
LINE 53: emergency contact one relationship
LINE 54: emergency contact one home number
LINE 55: emergency contact one cell number
LINE 56: emergency contact two first name
LINE 57: emergency contact two last name
LINE 58: emergency contact two relationship
LINE 59: emergency contact two home number
LINE 60: emergency contact two cell number
LINE 61: health factor one name
LINE 62: is health factor one life threatening
LINE 63: is health factor one plan of care required
LINE 64: is health factor one medications required
LINE 65: health factor two name
LINE 66: is health factor two life threatening
LINE 67: is health factor two plan of care required
LINE 68: is health factor two medications required
LINE 69: health factor three name
LINE 70: is health factor three life threatening
LINE 71: is health factor three plan of care required
LINE 72: is health factor three medications required
```

Figure 17. File format for student.txt



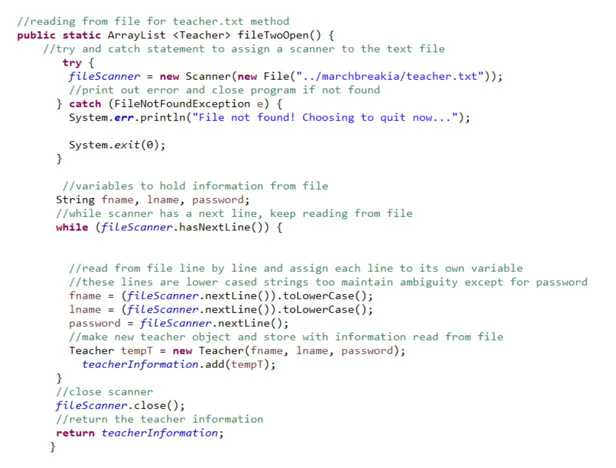
```
LINE 1: first name
LINE 2: last name
LINE 3: password
```

Figure 18. File format for teacher.txt

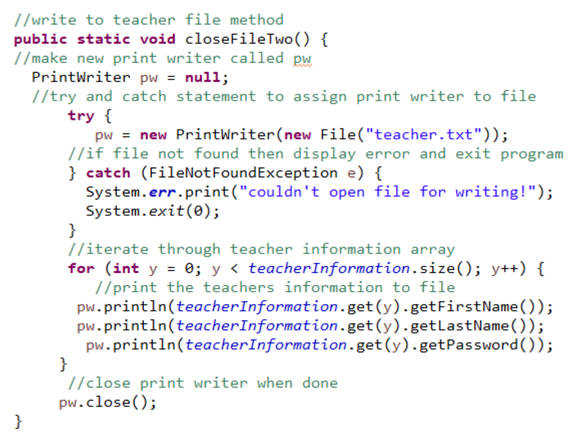


```
import java.io.PrintWriter;
import java.util.Scanner;
```

Figure 19. Libraries used in Menu.java class



```java
//reading from file for teacher.txt method
public static ArrayList <Teacher> fileTwoOpen() {
    //try and catch statement to assign a scanner to the text file
      try {
       fileScanner = new Scanner(new File("../marchbreakia/teacher.txt"));
       //print out error and close program if not found
     } catch (FileNotFoundException e) {
       System.err.println("File not found! Choosing to quit now...");

       System.exit(0);
     }

      //variables to hold information from file
     String fname, lname, password;
     //while scanner has a next line, keep reading from file
     while (fileScanner.hasNextLine()) {


       //read from file line by line and assign each line to its own variable
       //these lines are lower cased strings too maintain ambiguity except for password
       fname = (fileScanner.nextLine()).toLowerCase();
       lname = (fileScanner.nextLine()).toLowerCase();
       password = fileScanner.nextLine();
       //make new teacher object and store with information read from file
       Teacher tempT = new Teacher(fname, lname, password);
         teacherInformation.add(tempT);
     }
     //close scanner
     fileScanner.close();
     //return the teacher information
     return teacherInformation;
    }
```

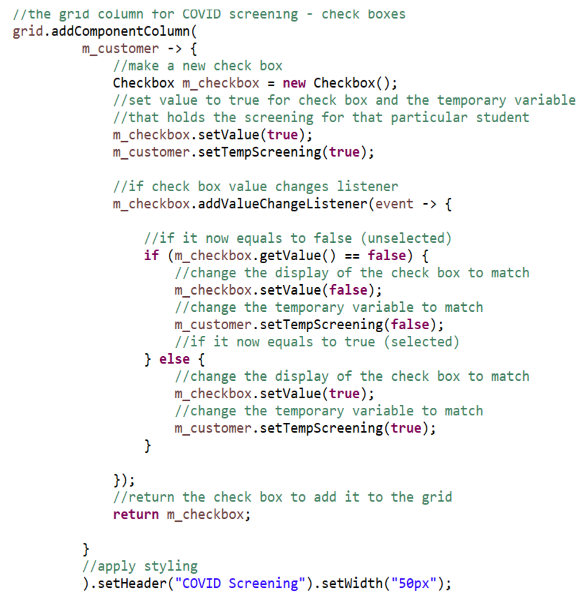


```java
//write to teacher file method
public static void closeFileTwo() {
//make new print writer called pw
  PrintWriter pw = null;
  //try and catch statement to assign print writer to file
      try {
         pw = new PrintWriter(new File("teacher.txt"));
      //if file not found then display error and exit program
      } catch (FileNotFoundException e) {
        System.err.print("couldn't open file for writing!");
        System.exit(0);
      }
      //iterate through teacher information array
      for (int y = 0; y < teacherInformation.size(); y++) {
         //print the teachers information to file
       pw.println(teacherInformation.get(y).getFirstName());
       pw.println(teacherInformation.get(y).getLastName());
        pw.println(teacherInformation.get(y).getPassword());
     }
      //close print writer when done
     pw.close();
}
```

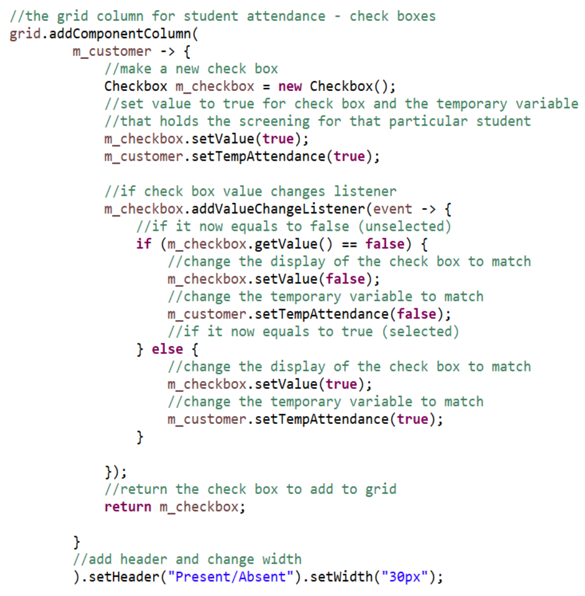
Figure 20. Reading and writing to file for teacher.txt

8: GUI considerations - Grid Display

For my user interface, I used a grid display[3] provided in the Vaadin Flow Java API. I modified it to better fit the client's requirements by adding inline editing to allow the client to add the appropriate reasons along with two columns of checkboxes for attendance and COVID screenings instead of the pre-given text fields (Figure 20). I also binded the modifications made in the grid to the student's attributes through the use of a binder. These modifications provided an easier navigation process for the client (Figure 21).



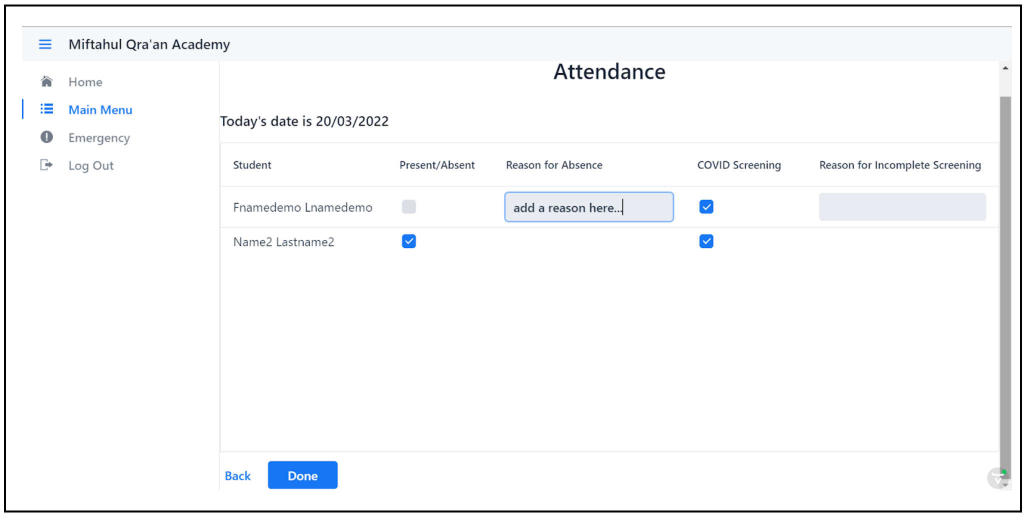
```java
//the grid column for COVID screening - check boxes
grid.addComponentColumn(
        m_customer -> {
            //make a new check box
            Checkbox m_checkbox = new Checkbox();
            //set value to true for check box and the temporary variable
            //that holds the screening for that particular student
            m_checkbox.setValue(true);
            m_customer.setTempScreening(true);

            //if check box value changes listener
            m_checkbox.addValueChangeListener(event -> {

                //if it now equals to false (unselected)
                if (m_checkbox.getValue() == false) {
                    //change the display of the check box to match
                    m_checkbox.setValue(false);
                    //change the temporary variable to match
                    m_customer.setTempScreening(false);
                    //if it now equals to true (selected)
                } else {
                    //change the display of the check box to match
                    m_checkbox.setValue(true);
                    //change the temporary variable to match
                    m_customer.setTempScreening(true);
                }

            });
            //return the check box to add it to the grid
            return m_checkbox;

        }
        //apply styling
        ).setHeader("COVID Screening").setWidth("50px");
```



```java
//the grid column for student attendance - check boxes
grid.addComponentColumn(
        m_customer -> {
            //make a new check box
            Checkbox m_checkbox = new Checkbox();
            //set value to true for check box and the temporary variable
            //that holds the screening for that particular student
            m_checkbox.setValue(true);
            m_customer.setTempAttendance(true);

            //if check box value changes listener
            m_checkbox.addValueChangeListener(event -> {
                //if it now equals to false (unselected)
                if (m_checkbox.getValue() == false) {
                    //change the display of the check box to match
                    m_checkbox.setValue(false);
                    //change the temporary variable to match
                    m_customer.setTempAttendance(false);
                    //if it now equals to true (selected)
                } else {
                    //change the display of the check box to match
                    m_checkbox.setValue(true);
                    //change the temporary variable to match
                    m_customer.setTempAttendance(true);
                }

            });
            //return the check box to add to grid
            return m_checkbox;

        }
        //add header and change width
        ).setHeader("Present/Absent").setWidth("30px");
```

Figure 20. Implementation of two checkbox columns

[3] Grid: Components: Design system: Vaadin Docs. Components | Design System | Vaadin Docs. (n.d.). Retrieved March 18, 2022, from https://vaadin.com/docs/latest/ds/components/grid



Attendance

Today's date is 20/03/2022

Student	Present/Absent	Reason for Absence	COVID Screening	Reason for Incomplete Screening
Fnamedemo Lnamedemo	☐	add a reason here...	✓	
Name2 Lastname2	✓		✓	

Figure 21. Grid display on user interface

Word Count: 1013