

Criteria C - Development

OOP

This program uses OOP. Many different classes were used as their objects were treated like real-life entities such as the receptionist class, pharmacist class, etc. This allowed for the different functionalities of the program to be divided up systematically within the classes. The use of OOP will also be useful if more functionalities are required to be added to the program later on.

Custom Sort using Comparator

In the second tab of the program, appointments are displayed. Three different buttons allow the user to sort the appointments alphabetically based on the patient or doctors name or sort the appointments based on its date. As Appointment objects are not simply integers or Strings, three custom sorts were used to order the appointments. ArrayLists were used to store the appointments as they are dynamic arrays allowing their size to be changed therefore the use of Collections.sort() was a good solution to sorting the arrays. The method was customized to sort Appointment objects. The image below displays the code used to sort the appointments alphabetically based on the names of the patients involved.

```
/**
 * Sorts all the appointments alphabetically based on the name of the patient involved in the appointment.
 *
 * @return the sorted list of appointments
 */
public ArrayList<Appointment> sortByPatient()
{
    ArrayList<Appointment> sortedPatientAppointment = appointments;
    Collections.sort(sortedPatientAppointment, new Comparator<Appointment>() {
        public int compare(Appointment a1, Appointment a2) {
            if ((a1.getPatient().getName()).
                equals(a2.getPatient().getName()))
                return 0;

            char[] a1Char = (a1.getPatient().getName()).toCharArray();
            char[] a2Char = (a2.getPatient().getName()).toCharArray();

            // Determine the shortest name.
            int length = a1Char.length;
            if (a1Char.length > a2Char.length) {
                length = a2Char.length;
            }

            for (int i = 0; i < length; i++) {
                if (a1Char[i] < a2Char[i])
                    return -1;
                else if (a1Char[i] > a2Char[i])
                    return 1;
            }

            // If all the letters of the two names are the same for the number of letters as in the shortest name,
            // indicate that the shorter word appears first alphabetically.
            return length == a1Char.length ? -1 : 1;
        }
    });
    return sortedPatientAppointment;
}
```

Custom Queue

The patient's history in the program is displayed as a list of the examinations performed in their past which included information regarding symptoms, diagnosis, treatment, etc. It may be important to refer to this medical history for a patient when discussing current symptoms and problems. This list of past examinations is stored as a queue in terms of the newer examinations being enqueued onto the queue last. As more recent examinations are being added to the patient's history, their old examinations become less relevant and therefore the use of queue easily allows the deletion of these older examination while simultaneously adding new examinations. A custom implementation was used as it ensures that the number of examinations displayed remains below or equal to the set limit.

```

/**
 * Adds new examination that is performed after this examination.
 *
 * @param examination the new examination
 */
public void enqueue(Examination examination) {
    dequeue();
    if (size == 0)
    {
        rear = head = examination;
        size++;
    }
    else {
        rear.setNext(examination);
        rear = examination;
        size++;
    }
}

/**
 * Removes the first examination performed in this patient record.
 */
private void dequeue() {
    if (size == GUI.maximumExaminationsDisplayed) {
        head = head.getNext();
        size--;
    }
}

```

File IO

As the program is opened, its data is loaded from the text files. The data is loaded as text from the files and new corresponding objects are created in order to store the data inside the program. Then, as the program is exited after its use, the data stored within the objects is transferred to text files. Storing the data in text files is used because the program would lose all of the data collected after exiting if it was not implemented. Therefore, the use of File IO was crucial in the development of this program. The code below shows how the data about doctors was loaded from a text file and objects were created in the program to store them.

```

/**
 * Loads information about all the doctors from a text file and constructs
 * appropriate Doctor instances.
 *
 * @throws IOException If an I/O error occurs
 */
private void loadDoctorData() throws IOException {
    String firstName;
    String lastName;
    String email;
    long number;
    String uniqueID;
    String input;
    String address;

    BufferedReader br = new BufferedReader(
        new FileReader(new File( pathname: "data/doctors.txt")));
    while ((input = br.readLine()) != null) {
        firstName = input;
        lastName = br.readLine();
        email = br.readLine();
        String num = br.readLine();
        System.out.print(num);
        number = Long.parseLong(num);
        address = (br.readLine());
        uniqueID = br.readLine();
        doctors.add(new Doctor(firstName, lastName, email, number,address, uniqueID));
    }
    br.close();
}

```

Catching Exceptions

When using the program, the user is asked for different inputs many times. Therefore, it was important to ensure that the user gave appropriately formatted data. If the user did input invalid information, the program threw an exception. This program catches the exception and displays an appropriate error message which indicates the user's mistake to them. Handling exceptions is very important in order to ensure that the user enters appropriate data that will continue to allow the program to run smoothly. Exceptions are also thrown manually in the program which allows the program to easily distinguish between the causes of the exceptions and subsequently provide more clearer error messages to the user.

```

addTestButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        try {
            if(testNameField.getText().trim().equals(""))
                throw new ArrayIndexOutOfBoundsException();
            manager.getLabTests().add(testNameField.getText());
            editPossibleLabTestsFrame.dispose();
            createEditPossibleTestsDisplay();
        } catch (ArrayIndexOutOfBoundsException e1) {
            editPossibleLabTestsFrame.setAlwaysOnTop(false);
            int result = JOptionPane.showConfirmDialog( parentComponent: null, EMPTY_TEXTFIELD_MESSAGE,
                ERROR, JOptionPane.DEFAULT_OPTION);
            if (result == JOptionPane.OK_OPTION) {
                editPossibleLabTestsFrame.setAlwaysOnTop(true);
            }
        }
    }
});

```

Inheritance

Using inheritance, the Task class was set as the superclass of the Appointment class. A doctor may have as many tasks as possible of which some may be simply organizing their workload, planning certain requirements for upcoming treatments, writing reports or attending appointments with patients. Therefore, appointments are also tasks. However, as they involve patients unlike other tasks, it was used as a subclass of the Task class. Thus, whenever an appointment was created, it was also added as a task of the doctor involved in the appointment.

```

public class Appointment extends Task{

    String appointmentType;
    Patient patient;

    /**
     * Constructs a new Appointment instance using the specified information.
     *
     * @param purpose the purpose of the appointment
     * @param description the description of what will occur during the appointment
     * @param appointmentType the type of appointment
     * @param doctor the doctor involved in the appointment
     * @param patient the patient involved in the appointment
     * @param date the date of the appointment
     * @param startTime the starting time of the appointment
     * @param endTime the estimated ending time of the appointment
     */
    public Appointment(String purpose, String description, String appointmentType, Doctor doctor, Patient patient,
                       int date, int startTime, int endTime) {
        super(purpose, description, doctor, date, startTime, endTime);
        this.appointmentType = appointmentType;
        this.patient = patient;
    }

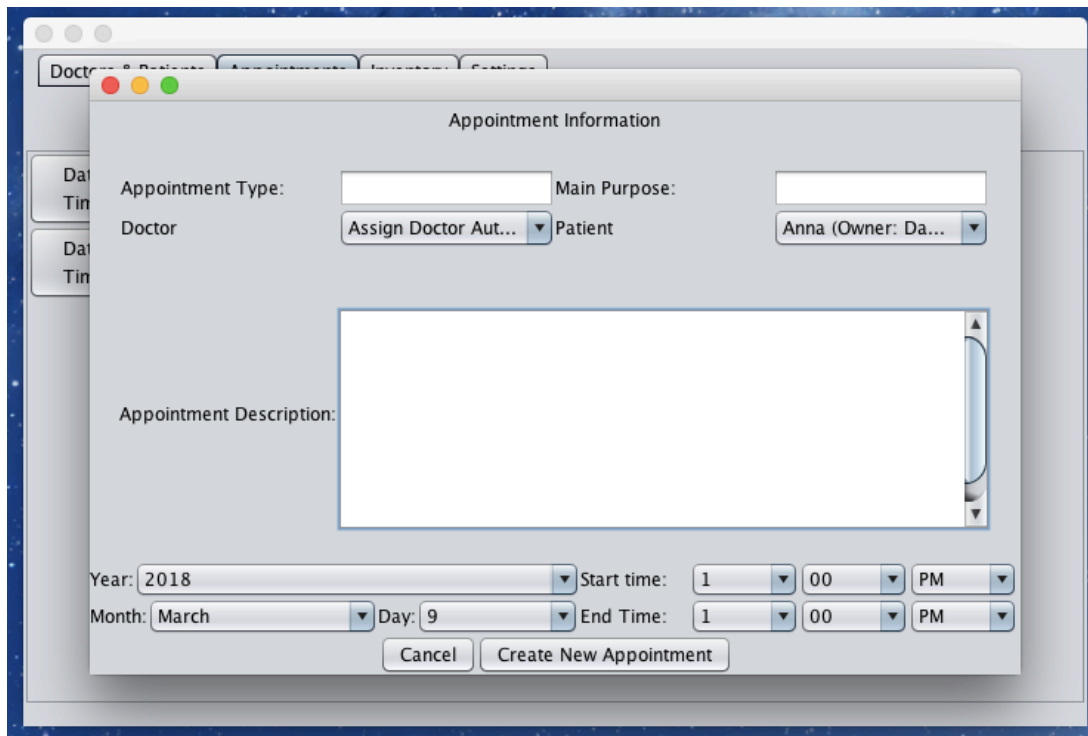
    /**
     * Returns the patient involved in this appointment.
     *
     * @return the appointment involved in this appointment
     */
    public Patient getPatient() {
        return patient;
    }

    /**
     * Returns the type of this appointment.
     *
     * @return the type of this appointment
     */
    public String getAppointmentType() {
        return appointmentType;
    }
}

```

GUI

This program was designed to be used by a relatively inexperienced computer user (my client is quite unfamiliar with using computers) thus by using GUI, it became more user-friendly and easier to learn. JFrames were used along with JPanels added inside the different layout managers in order to make the displays appear more professional. Multiple windows were displayed at once which caused the potential problem of the user attempting to click on both windows and use them at once. This resulted in a lot of exceptions being thrown in the program therefore, in the GUI, only one window was enabled if multiple were open.



The image shows a screenshot of a software application window titled "Appointment Information". The window is a modal dialog box with a title bar containing standard macOS window controls (red, yellow, green buttons). The background window is partially visible, showing tabs for "Doctors", "Patients", "Appointments", "Inventory", and "Settings".

The "Appointment Information" dialog box contains the following fields and controls:

- Appointment Type:** A text input field.
- Main Purpose:** A text input field.
- Doctor:** A dropdown menu with the selected option "Assign Doctor Aut..." and a small downward arrow.
- Patient:** A dropdown menu with the selected option "Anna (Owner: Da..." and a small downward arrow.
- Appointment Description:** A large, empty text area with a vertical scrollbar on the right side.
- Year:** A dropdown menu with "2018" selected.
- Month:** A dropdown menu with "March" selected.
- Day:** A dropdown menu with "9" selected.
- Start time:** A time selection control with three dropdowns: "1", "00", and "PM".
- End Time:** A time selection control with three dropdowns: "1", "00", and "PM".
- Buttons:** At the bottom, there are two buttons: "Cancel" and "Create New Appointment".

Word Count: 751