Section 1 – Features

The program first requires the user to choose which CSV file located in the "data/" directory (same directory level as "src/") they would like to load from a dropdown menu. This immediately shows the table with all the data. There is a search button which searches for an exact match, and this also has a reset button to show all of the data again. The default sort is by ascending ID, but this can be changed to any combination of ascending and descending with the column names from the dropdown menu – the reset button sets it back to the default.

The search and sort can also be done simultaneously, but it has to be in the order search then sort. The user can view a bar chart based on the frequency of values in a particular column – this only works for "BIRTHDATE", "DEATHDATE", "SSN", "SUFFIX", "ZIP" – as well as for a patient's age which is calculated based on birth date and death date. The graphs generated will only display data based on the results of what is in the search bar (empty means all the data will be shown).

The user can also download a JSON file of the data currently displayed – even after both searching and sorting, and after row operations. The user will be prompted where to save the file and for the filename. The user can also edit any row, changing the values for that row's column except for the ID), delete any row, or add a new row (a unique ID is automatically generated). All of these changes will be reflected in the patientList page. By default, every row operation (edit, delete, or add new row) results in a new CSV file being created – the filename will be the current filename with the current timestamp appended, and it is saved into the "data/" directory; the operations persist until the user goes back to the home page and selects a different file. When adding a new row, there is a "clear" button which clears all fields except the ID field. There is a similar "revert" button when editing a row, which resets the values to what they were before editing that row began.

Section 2 – Describing & Evaluating my Design & Programming Process

I tackled the coursework in quite a straightforward fashion – I went through the requirements in order, as this gave me a good idea of what to work on during the next stage of the development process. I began by simply writing the classes with the methods described in the first couple of requirements, but when it came to writing the DataLoader class, I knew that I really only required a single method, but I made that method a static class because I realised that the only time that I would use it is when the model needs to load the data into a DataFrame. At this point, I thought that it would be better to read ahead and have a general idea of how I would like to design my code.

I included a DataFrame attribute as part of the Model – though this could have been public static due to the use of the Singleton design pattern required for this project, I did not want to give the servlets direct access to the actual data, so I simply opted for a private instance attribute as this wouldn't make much of a difference due to the Singleton framework. Additionally, I created a ModelFactory class which would have a public static getModel() method to allow access to get the private static Model attribute within the ModelFactory, since only 1 Model would be required. I also decided to overload the public getModel() method. This is because, initially, when the ModelFactory's method has just been called, it would need a filename to populate the DataFrame, however, until the user goes back to the home page, they should be working on and editing the same file, so the filename argument doesn't need to be passed in every time – hence an overloaded method for when the filename is required and for when it is not. Of course, when a new file is being worked on, the filename argument is used.

Since the Controllers (servlets) required data to send to the Views (JSPs), I decided to implement the required methods in the Model class in order to preserve the MVC pattern. Any methods that were needed by the servlets I made public, while the helper methods within the Model class that would only need to be used within it were set to private, for example, the getAge() method which is only used in the sort() method when sorting by age or in the getBarChart() method for the actual ages to be counted.

For the servlets, I realised that most of the servlets were centred around the patientList page, and a lot of code was being repeated in those servlets, for example in the SearchServlet, SortServlet, and others, since they were all forwarded back to the patientList.jsp file. A lot of the attributes also always had to be set and passed in, and most of the time these were the same, so I decided it would be better to create an abstract class called AbstractPatientsServlet and let the other servlets be subclasses of this class and just use its methods – I set them to protected since they would need to be used by the subclasses but no other classes. Additionally, I created another abstract class that inherited from this, AbstractPatientsFeaturesServlet, as DeleteRecordServlet, EditRecordServlet, and AddRecordServlet all had just slightly more features and attributes to set than the other servlets inheriting from AbstractPatientsServlet, so I was able to abstract those and put them in this abstract class, overriding some methods from AbstractPatientsServlet to be more specialised for those 3 servlets. After doing this, I also found out that my SearchServlet and SortServlet had the exact same functionality, the only difference being the exact web routes / get requests they were handling, so I simply combined to 2 servlets into a SearchAndSortServlet, using the fact that a single servlet could handle multiple routes.

In regard to the JSPs, I tried to modularise as much as possible to make partials, splitting up HTML sections where they had to be used multiple times and putting them into separate JSPs, including them where necessary. By doing this, I significantly reduced the amount of redundant code. However, I think my use of CSS could have been much better – there were many times where I simply used a style attribute in a HTML tag instead of actually using the styles.css file I had created – this could make the project quite hard to manage if it were to be expanded upon.

Though I think I designed the structure of my files and code quite well, I certainly could have improved in a couple of aspects. For one, I used a lot of `return null` statements, which is not good practice. Instead, I could have asserted not null, thrown an exception, or, probably the best option, used Optional values using Optional.of(), Optional.empty(), and Optional.ofNullable(). Additionally, though I have a page that is supposed to display all errors, there are some particular cases where the default Apache Tomcat error page is displayed. Fortunately, the website doesn't crash because I still have exception handling, however, I think that I potentially have too many exceptions and try-catch blocks – perhaps some more designing in the initial stages of the project could have helped me avoid this and handle the exceptions better.

Overall, I did quite well for this coursework – I met all the requirements, learned a lot about Java and OOP, and I managed to implement additional functionality, integrating JavaScript, HTML, CSS, and Java. I also think I had good use of abstraction with the abstract classes and inheritance, but perhaps there were some places, such as exception handling, where I could have improved. I also could have used other OOP design strategies more effectively, such as interfaces, however I did not identify where I could use these in my initial development phases, which taught me the importance of understanding exactly what requirements need to be implemented and planning ahead. Additionally, though it was not a strict requirement, I would have liked to make the website look slightly more aesthetically pleasing.