Criteria C: Development

Word count: 809

Techniques:

- Use of a Graphic output
- Use of ListView and ObservableList
- Use of user input
- Use of error checking using user-inputted bounds

Use of a Graphical Output:

My program has a GUI display of the planner that updates in real time through a graphics window. The window allows you to add tasks using text inputs for each variable, which are listed on the screen. You can also remove the tasks on the list by clicking the task and then hitting the remove button. Most importantly you can sort the list by priority by pushing a button which then sorts the list based on the algorithm I made.

```
// Task List View
taskListView = new ListView<>();
taskListView.setPrefWidth(600);
taskListView.setPrefHeight(400);
taskListView.setCellFactory(param -> new ListCell<Task>() {
    @Override
  protected void updateItem(Task item, boolean empty) {
      super.updateItem(item, empty);
      if (empty || item == null) {
          setText(null);
      } else {
          setText(item.toString());
      }
  }
});
borderPane.setCenter(taskListView);
```

The code above is an example of how my ListView is set up. It first initializes and set defines the height and width of the list. Then a cell factory is created which defines how the cells are created Then the updateItem method takes two parameters, a new item for the cell and a boolean called “empty” to indicate whether the cell is empty or not. The “super.updateItem(item, empty)” line ensures that the superclass implementation of updateItem is also called. Finally, it checks if the cell is empty or not, to ensure that the cell information is always accurate as items are removed or added. This piece of code initializes the list and makes sure that it's updated correctly based on changes in the data.

Use of ListView and ObservableList :

I needed a way to display a list for my tasks while being able to store its data, because of that I used a ListView in combination with an ObservalbeList. The ListView displays the tasks in a list to the user with each item in the ListView corresponding to a task. An observable list is a special type of list in the JavaFX interface that extends on a basic list by notifying any observers like UI elements about the change. I use these two in combination to manage and display each task in the UI while making sure data and the UI are synchronized. This works by modifying the underlying ObservalbleList which the Listview automatically updates from. So any modification of the ObservableList like sorting or deleting items in the list will be shown on the UI with ListView. ListView and ObservableList are declared together as a class member with ListView internally managing an ObservableList which stores and manages items.

```
private void deleteSelectedTask() {
    ObservableList<Task> selectedItems = taskListView.getSelectionModel().getSelectedItems();
    taskListView.getItems().removeAll(selectedItems);
}

private void sortTasksByPriority() {
    ObservableList<Task> items = taskListView.getItems();
    items.sort(Comparator.comparingInt(Task::getPriority).reversed());
    taskListView.setItems(items);
}
```

Here is an example of the ListView and ObservalbeList being used. In the deleteSelectedTask method, the task is being removed from the ObservableList which also removes the ListView which controls UI. Similarly, the sortTasksByPriority method retrieves and sorts the underlying ObservableList. Since the ListView is observing the ObservableList it automatically updates and displays the task in order.

Use of user input:

I need to be able to get the user's input so I use text fields to help me gather the data. This data is key as the main focus of the program and helps me determine the priority of each assignment from using the data.

Use of error checking using user-inputted bounds:

There were too many ways a user could input something incorrectly and break the program. Therefore this error checking was needed to keep things running smoothly. The picture below shows my error-checking code:

```
        try {
            points = Integer.parseInt(pointsStr);
            time = Integer.parseInt(timeStr);
        } catch (NumberFormatException e) {
            // Handle invalid input for points or time
            System.err.println("Invalid input for points or time. Please enter
valid integers.");
            return;
        }
```

This error check occurs when a user inputs an incorrect value in any of the text boxes. If not for this error checking the user could input a character where an integer is supposed to go, breaking the code. I prevented this by using a try-catch statement which catches the NumberFormatException error that would otherwise occur if the character or string was inputted where an integer is supposed to go.

Use of algorithm:

When deciding how I would sort the tasks by priority I came up with this algorithm:
First I decided to weigh each category as it allows me to adjust how much each section is worth easily:
Number of points = 2
time required = 2
Due Date = 3
I made the due date worth more as that was the most important.
For a test, I add 100 points to the task so it goes to the top of the list.

To calculate how much each section is worth I did some simple calculations:
Points would be:
Number of points the assignment was worth * the weight (2)
Time required would be: 1 / time * weight (2) * 60
Due date would be: 1 / the days until due + 1 * weight (3)

I chose for time required and due date to be inversely proportional as the priority for those should exponentially increase to show how you are running out of time to do the assignment.

Finally, I added up the totals for each variable to calculate the assignment's priority number, which then got sorted by highest to lowest, with the highest being the most important task to do.