



Assignment Cover Letter (Individual Work)

Student Information:	Surname	Given Names	Student ID Number
1.	Lie	Karsten	2440035463

Course Code	: COMP6699	Course Name	: Object Oriented Programming
Class	: L2AC	Name of Lecturer(s)	: Jude Joseph Lamug Martinez
Major	: Computer Science		
Title of Assignment (if any)	: To Do Manager		
Type of Assignment	: Final Project		
Submission Pattern			
Due Date	: 21-06-2021	Submission Date	:

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer's instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

Plagiarism/Cheating

BiNus International seriously regards all forms of plagiarism, cheating and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

Declaration of Originality

By signing this assignment, I/we* understand, accept and consent to BiNus International terms and policy on plagiarism. Herewith I/we* declare that the work contained in this assignment is my/our* own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student:

(Name of Student)

Karsten Eugene Lie

Table of Contents

1. INTRODUCTION	3
2. PROJECT SPECIFICATIONS	3
2.1 Aim	3
2.2 Apps, Frameworks, Libraries, etc.	3
2.3 Languages	3
2.4 Features	3
3. UML DIAGRAM	4
4. HOW IT WORKS	5
4.1 First Start Up	5
4.2 Adding Tasks	6
4.3 1 Task Added	8
4.4 Multiple Tasks	9
4.5 Delete Task	10
5. CODE	11
5.1 Navigation.java	11
5.2 ToDoData.java	12
5.3 TaskController.java	14
5.4 TaskCreationController.java	16
6. PROJECT GITHUB LINK:	18
7. CREDITS:	18

Project Name:
To Do Manager

Name:
Karsten Eugene Lie

ID:
2440035463

1. Introduction

To Do Manager is a simple To Do GUI where users can input their tasks. This program can be used by everyone who thinks that they need a To Do list on their computer and wants a simple and clean design for it with all the basic functionalities of a To Do list.

2. Project Specifications

2.1 Aim

The aim of this project is the same as a normal to do list, reminding people of what to do in their day-to-day life, but just on a computer. This could be more beneficial for people who do most of their tasks on their computer so that they can keep the program open to always remind themselves of what they need to do. This program would also help with time management as users can decide when they want to do their tasks. There are also indications if the task is important so that users can prioritize those first.

2.2 Apps, Frameworks, Libraries, etc.

- Java (ver. 16.0.1)
- JavaFX (ver. 16.0.0)
- Scene Builder (ver. 16.0.0)

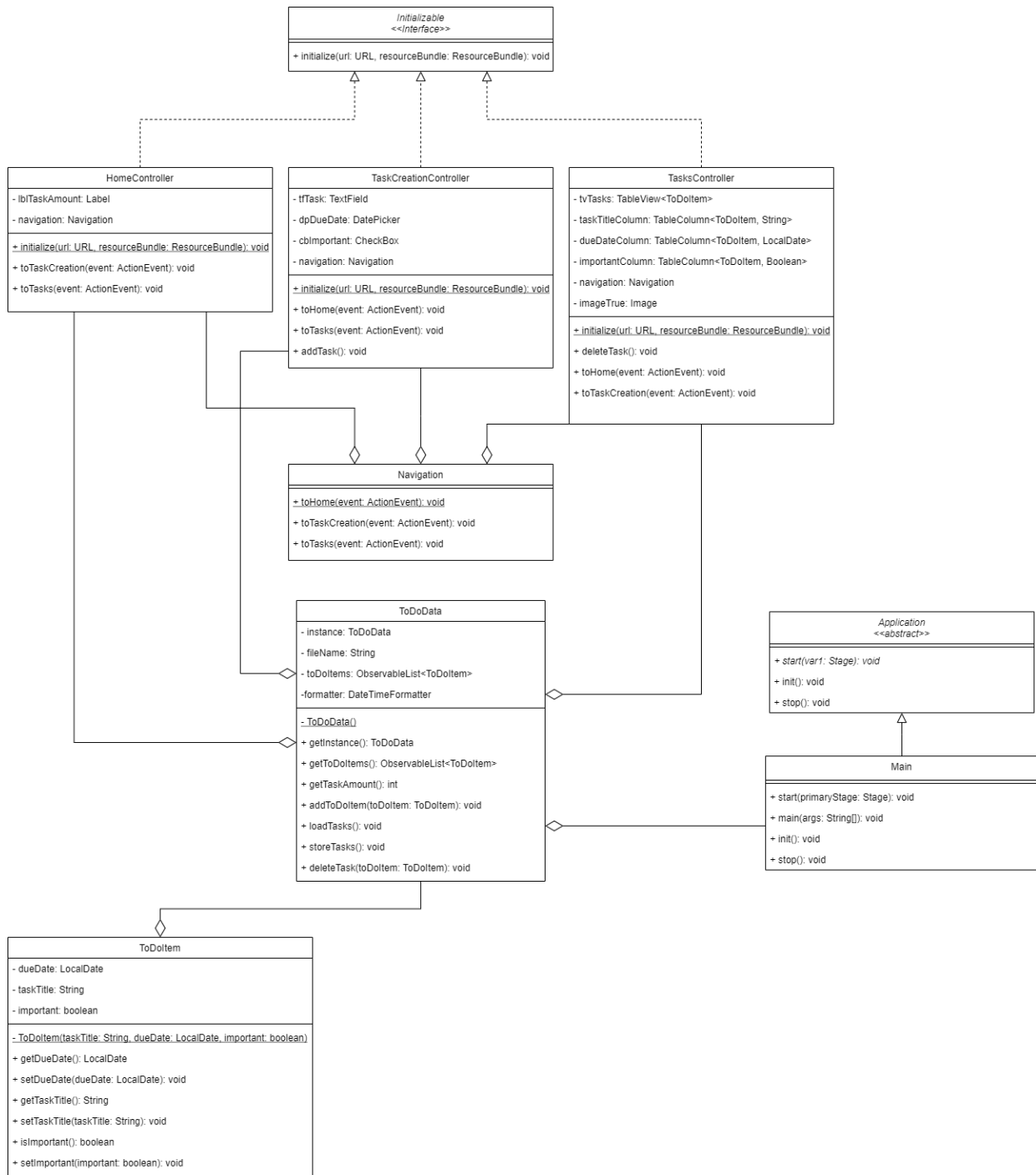
2.3 Languages

- Java
- CSS

2.4 Features

- Add tasks
- Delete tasks
- Displaying total tasks on home page
- Sort tasks by the due date, importance, or alphabetically (default of table view)
- Tasks are stored on the local computer (inside txt file)

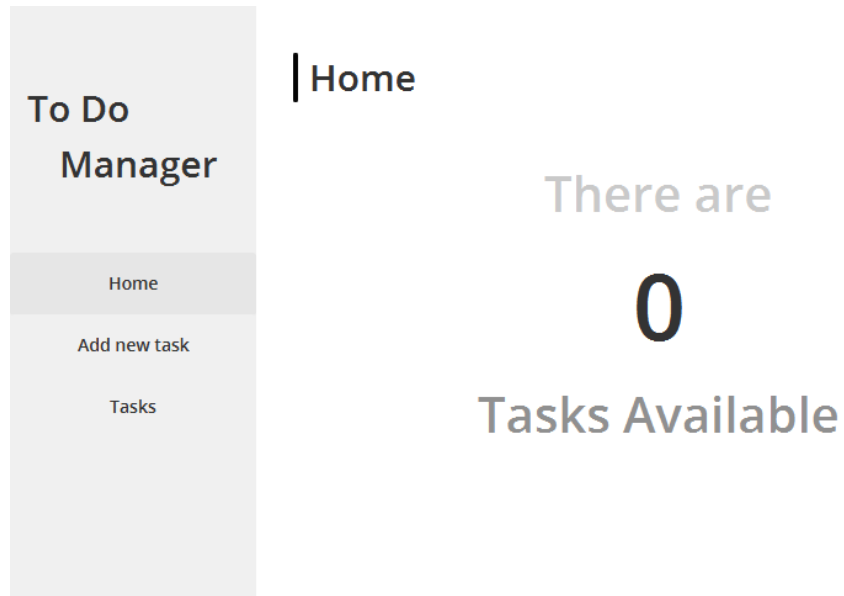
3. UML Diagram



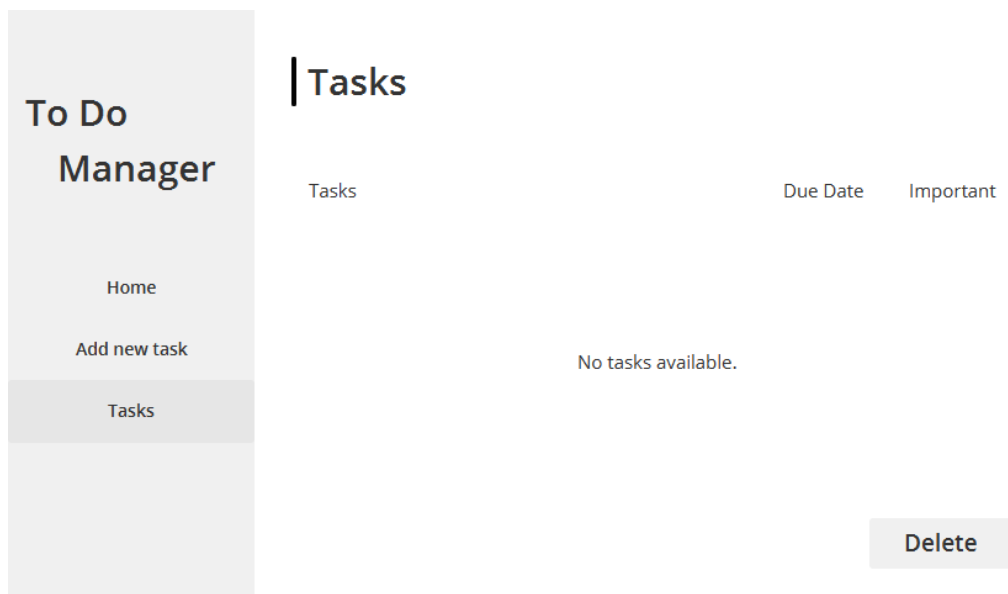
The diagram above shows the UML diagram of the To Do Manager. We can see that all the controller classes implement the Initializable class. This class is used to initialize all the labels or texts to display as the scene is for that specific controller is opened. The controller classes also aggregate from the Navigation class. This class is created to prevent long duplicate codes and it is used to navigate between the tabs or scenes. There are also a lot of classes that aggregate the ToDoData class. This class is useful as it has the `getInstance()` method to access the other methods inside the class. Of course, the **ToDoData** class also aggregates the **ToDoItem** class as that is where new Tasks are created. After tasks are created, they are transferred to the **ToDoData** class to be added to a list.

4. How It Works

4.1 First Start Up



When the user first launches the GUI, they are introduced with the home screen that displays “There are 0 Tasks Available”. The digit will change depending on the number of tasks the user has made. From here, the user can go to the sidebar and click on “Add new task” or “Tasks”.



This is how the “Tasks” view will look like when there are no tasks added yet. In the middle of the table, a text saying, “No tasks available.” will be displayed.

4.2 Adding Tasks

To Do Manager

Home

Add new task

Tasks

Task Creation

Task

I need to...

Due Date

Pick a date...

☐ Important

Add

To add tasks users can click the “Add new task” button and they will be greeted with this screen. Here, a text field, a date picker, and a checkbox will be available for the user.

To Do Manager

Home

Add new task

Tasks

Task Creation

Task

OOP Project

Due Date

Pick a date...

☐ Important

Add

In the text field, the user can input their task title. In this example, I used OOP Project

To Do Manager

Home

Add new task

Tasks

Task Creation

Task

OOP Project

Due Date

< June >

< 2021 >

Sun	Mon	Tue	Wed	Thu	Fri	Sat
30	31	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	1	2	3
4	5	6	7	8	9	10

☐ Important

Next is the date picker. This is where users can pick their due date. The date picker also disabled the dates before today as it does not make sense for the due date to be on a day before the current day.

To Do Manager

Home

Add new task

Tasks

Task Creation

Task

OOP Project

Due Date

6/21/2021

☒ Important

Add

Once the user has chosen a date for the due date, the user can then check or leave the checkbox that says “Important”. This important checkbox is for tasks that the user thinks are important, and it will be marked on the “Tasks” tab later.

To Do Manager

Home

Add new task

Tasks

Task Creation

Task

OOP Project

Due Date

6/21/2021

☒ Important

Add

After the user filled everything correctly, the user can click the “Add” button. The button was initially grey, but once the user hovers on it, it will turn green. After clicking the button, the task will be added to the “Tasks” tab and a success message will appear on the top right. The program will also clear the user inputs to prepare for the next input.

4.3 1 Task Added

To Do Manager

Home

Add new task

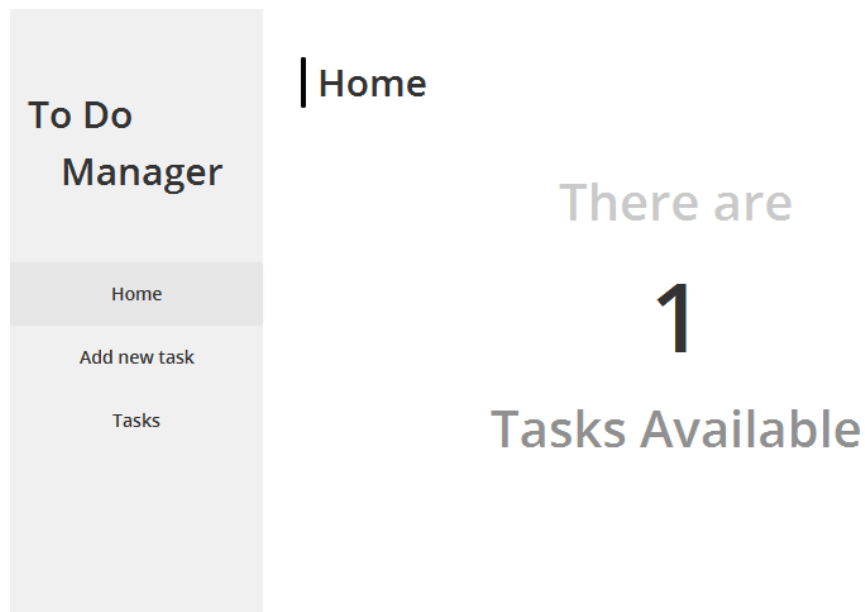
Tasks

Tasks

Tasks	Due Date	Important
OOP Project	21 June	✓

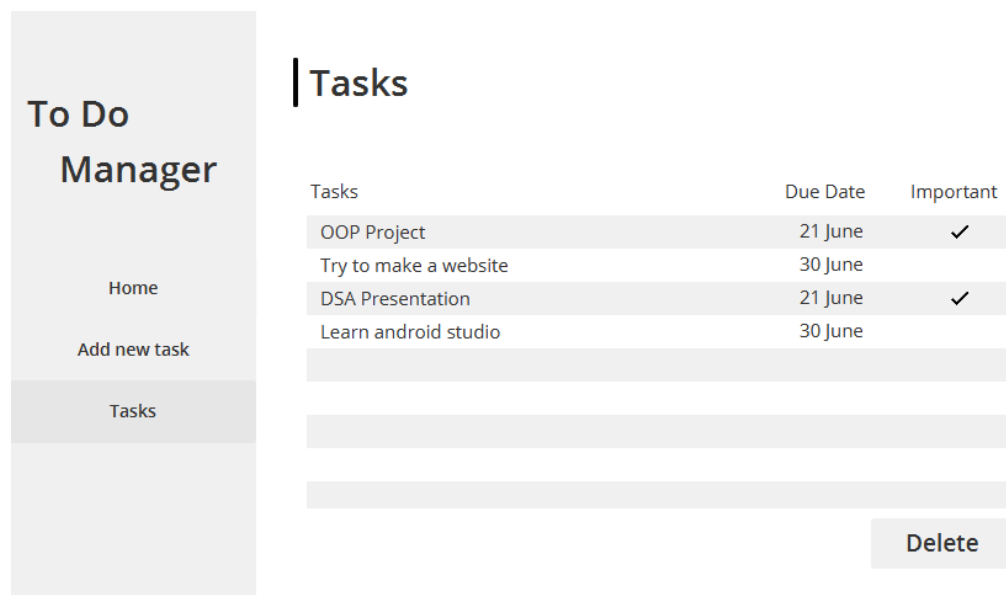
Delete

Once a task is added, it will show up on the “Tasks” tab in a table view, where users can see the “Tasks” column, which is for the task name, the “Due Date” column, which shows the due date, and the “Important” column, which shows a tick if the task is important. If the task is not important, nothing will show up on that column.

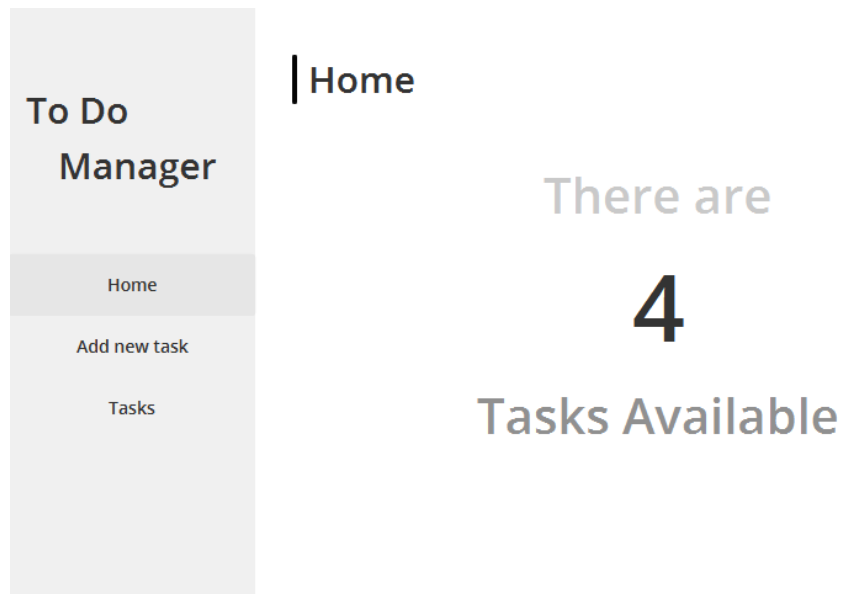


Other than the “Tasks” tab, the “Home” tab will also update. Since the user added 1 task, it will display that there is 1 task available.

4.4 Multiple Tasks

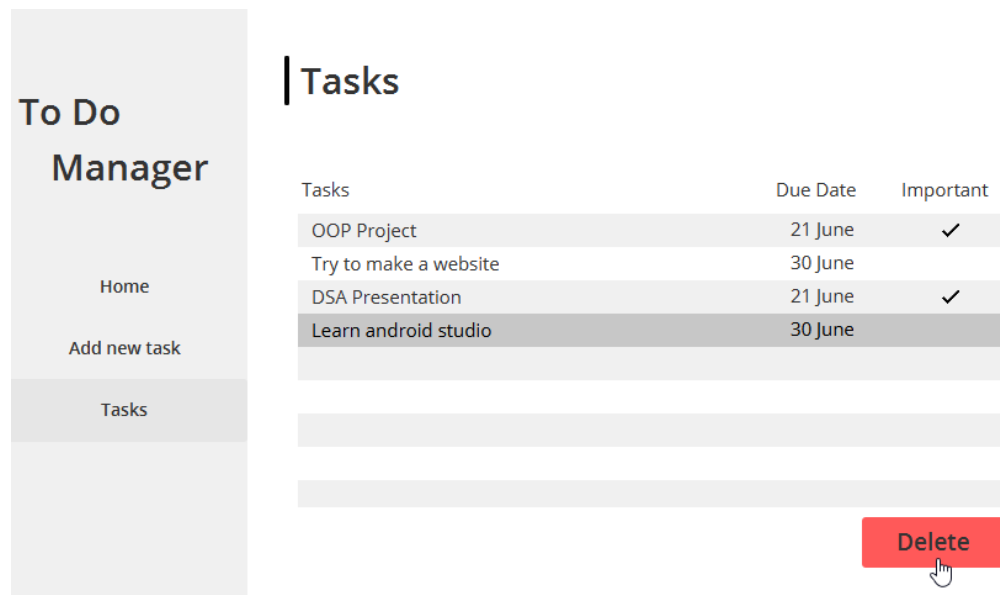


If the user has added a few other tasks, this would be how it would look in the “Tasks” tab. We can see the differentiation between tasks that are important and tasks that are not. The default order for the table is by the latest task added. The top being the first task added while the bottom is the last task added. The user can sort it by task name or due date or importance just by clicking on one of the column headers.

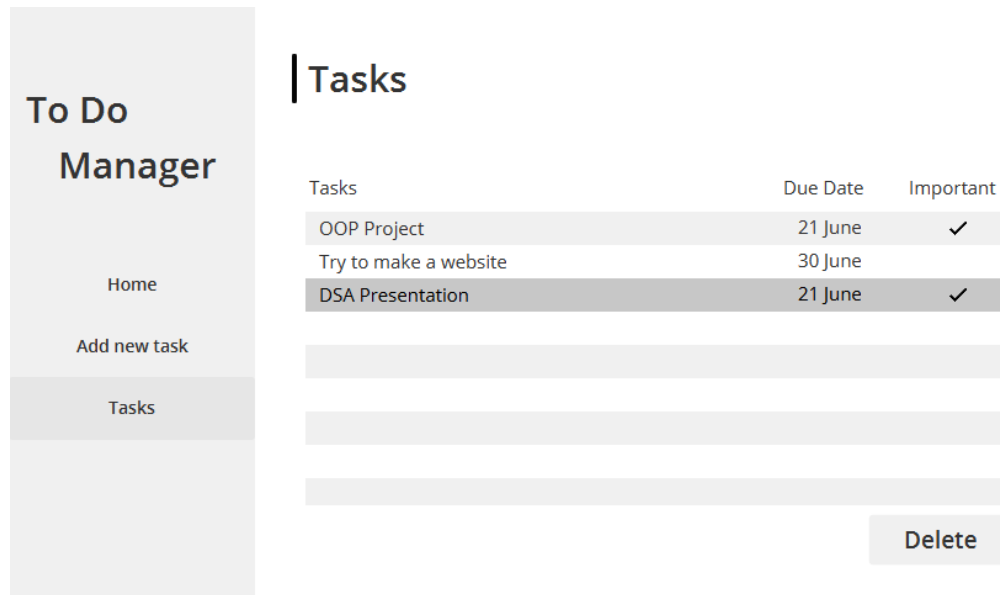


Again, the number of tasks will also be updated in the “Home” tab depending on how many tasks the user added.

4.5 Delete Task



In the “Tasks” tab, the user also has the option to delete a certain task. All the user needs to do is click on one of the tasks and it will be highlighted by a darker grey highlight. Then the user can go to the delete button and click on it.



After the user presses delete, the task will instantly be removed from the table. And of course, the “Home” tab will also update the number of tasks available.

5. Code

The codes displayed in this section will be codes that are not self-explanatory, in other words, codes that are a little complicated, or codes that are, what I think, are quite fascinating to look at.

5.1 Navigation.java

```

17 public class Navigation {
18
19     // Method to change to the Home scene
20     @ public void toHome(ActionEvent event) throws IOException {
21         URL url = new File( pathname: "src/main/fxml/Home.fxml").toURI().toURL(); // Gets the Home view URL
22         Parent root = FXMLLoader.load(url); // Loads the Home view
23         Stage stage = (Stage) ((Node) event.getSource()).getScene().getWindow(); // Sets the new stage
24
25         // Sets the new scene with the Home view
26         Scene scene = new Scene(root);
27         stage.setScene(scene);
28
29         stage.show(); // Show the new display
30     }

```

The Navigation class is a custom class that was aimed to prevent long duplicate codes inside the controllers. The methods in this class are basically to change scenes depending on what the user clicks (Home, Add new task, Tasks). First, it gets the URL of the FXML file, and it gets passed to the root. The method will then prepare a new stage and a new scene, where the root is passed in. Finally, it displays the stage.

5.2 ToDoData.java

```
57 // Method to load tasks from the text file
58 public void loadTasks() throws IOException {
59     toDoItems = FXCollections.observableArrayList();
60     Path path = Paths.get(fileName); // Gets the path of the text file
61     BufferedReader br = Files.newBufferedReader(path); // Open the text file using a Buffered Reader
62
63     String input;
64
65     try {
66         while ((input = br.readLine()) != null) {
67             String[] itemPieces = input.split(regex: ","); // Splits each column using commas
68
69             String taskTitle = itemPieces[0]; // Task title for the first column
70             String dueDateString = itemPieces[1]; // Due date for the second column
71             String importantString = itemPieces[2]; // A true or false of whether it is important for third column
72
73             LocalDate dueDate = LocalDate.parse(dueDateString, formatter); // Reads the date using the formatter
74             boolean important = Boolean.parseBoolean(importantString); // Reads the boolean value for important
75             ToDoItem toDoItem = new ToDoItem(taskTitle, dueDate, important); // Makes a new ToDoItem using constructor
76             toDoItems.add(toDoItem); // Loads the item into the program
77         }
78     } finally {
79         if(br != null) {
80             br.close(); // Once there is no more lines, close the reader
81         }
82     }
83 }
```

ToDoData is a custom-made class, and its aim was to deal with the processes of the ToDoItem (which is also a custom-made class). However, the main method that I will be displaying is **loadTasks()** and **storeTasks()**.

The screenshot above shows **loadTasks()**, which is a method to load tasks from previous sessions. This means when a user adds a few tasks and closes the program, the tasks will still be available when the user opens the program again. This method is possible by making a new instance of Buffered Reader and reading the “Tasks.txt” file. Inside the text file, each line contains 1 task, and each task is separated into 3 columns using a comma (.). Once it finishes reading all the lines, it will add it to the list of tasks and close the text file.

```

84
85 // Method to store tasks into the text file
86 public void storeTasks() throws IOException {
87     Path path = Paths.get(fileName); // Gets the path of the text file
88     BufferedWriter bw = Files.newBufferedWriter(path); // Open the text file using a Buffered Writer
89
90     String input;
91
92     try {
93         Iterator<ToDoItem> iterator = toDoItems.iterator(); // Declare an iterator for the ToDoItems list
94
95         // While loop to get the tasks from the list
96         while (iterator.hasNext()) {
97             ToDoItem toDoItem = iterator.next();
98             bw.write(String.format("%s,%s,%s", // Writes into the text file by separating each column with commas
99                 toDoItem.getTaskTitle(),
100                 toDoItem.getDueDate().format(formatter),
101                 toDoItem.isImportant()));
102             bw.newLine(); // Write each item on a new line
103         }
104     } finally {
105         if (bw != null) {
106             bw.close(); // Once there is no more lines, close the writer
107         }
108     }
109 }
110

```

The code above is the **storeTasks()** method. This code does the opposite of **loadTasks()**. **loadTasks()** loads the task items every time the program starts, while **storeTasks()** stores the task items every time the program closes. To store tasks, I used Buffered Writer to write into the text file. First, it will create an iterator for the task items. Then it will iterate through each task and get their title, due date, and “important” variable and write into the text file, separating each variable with a comma. After that, it will move on to the next item. Once there are no items left, the text file will be closed.

Both **storeTasks()** and **loadTasks()** are called in the Main class so that it could be called whenever the program starts or closes

5.3 TaskController.java

```
44 // Method to initialize a certain functionality inside the scene
45 @Override
46 public void initialize(URL url, ResourceBundle resourceBundle) {
47     // Set the "taskTitle" variables inside the Tasks column
48     taskTitleColumn.setCellValueFactory(new PropertyValueFactory<>("taskTitle"));
49
50     DateTimeFormatter formatter = DateTimeFormatter.ofPattern("d MMMM"); // Declare a formatter for Date
51
52     // Lambda expression to change the format of the Date inside the Due Date column using the formatter
53     dueDateColumn.setCellFactory(column -> updateItem(date, empty) -> {
54         super.updateItem(date, empty);
55         if (empty) {
56             setText("");
57         } else {
58             setText(formatter.format(date));
59         }
60     });
61
62 // Set the "dueDate" variables inside the Due Date column
63 dueDateColumn.setCellValueFactory(new PropertyValueFactory<ToDoItem, LocalDate>("dueDate"));
64
65 // Lambda expression to change the boolean texts to be images or null
66 importantColumn.setCellFactory(column -> new TableCell<ToDoItem, Boolean>() {
67     private final ImageView imageView = new ImageView();
68
69     {
70         // Sets the image to be 20 by 20
71         imageView.setFitWidth(20);
72     }
73 }
```

Heading into the controllers, let us first start with the TaskController. This class is where the table view is configured. We can see that there is overriding on the *initialize* method. This is because I implemented the *Initializable* interface, and this interface is used for all the controllers inside this program. What the *initialize* method does is basically set up the controls (ex. Label, TableView, Button) whenever the scene of the controller loads.

In this case, we initialized the TableView for our tasks table. We can see codes there that uses **setCellValueFactory(new PropertyValueFactory<>())**. This method is to direct the variables into the right column. For example, *taskTitle* variable into the *taskTitleColumn*.

There is another method, **setCellFactory()**, which is usually followed with a lambda expression. This method is used to change how the data is displayed on the cell. For example, in the case of the *dueDateColumn*, I used that method to change the format of the date so that it is easier to read.

```

68 // Lambda expression to change the boolean texts to be images or null
69 importantColumn.setCellFactory(column -> new TableCell<ToDoItem, Boolean>() {
70     private final ImageView imageView = new ImageView();
71
72     {
73         // Sets the image to be 20 by 20
74         imageView.setFitWidth(20);
75         imageView.setFitHeight(20);
76         setGraphic(imageView);
77     }
78
79     // Method to change the text "true" to a tick icon and "false" to nothing
80     @Override
81     protected void updateItem(Boolean item, boolean empty) {
82         if (empty || item == null) {
83             imageView.setImage(null);
84         } else {
85             imageView.setImage(item ? imageTrue : null);
86         }
87     }
88 });
89
90 // Set the "important" variables inside the Important column
91 importantColumn.setCellValueFactory(new PropertyValueFactory<ToDoItem, Boolean>(s: "important"));
92 tvTasks.setItems(ToDoData.getInstance().getToDoItems()); // Gets the items from the list from ToDoData class
93 }
94

```

In the same controller, we can see that I also used the **setCellFactory()** method on *importantColumn*. This is used to change the Boolean values into images or nothing. So, if a value in the important column is true, the value will change to a tick image. If it is false, it will be changed to blank.

Once I configured all the cells and directed all the variables to the right column, I add all the tasks that are in the tasks list into the tasks table by using the **setItems()** method along with **ToDoData.getInstance().getToDoItems()** as the argument.

5.4 TaskCreationController.java

```
21 public class TaskCreationController implements Initializable {
22
23     // Injecting FXML to connect some controls with the controller
24     @FXML
25     private TextField tfTask;
26     @FXML
27     private DatePicker dpDueDate;
28     @FXML
29     private CheckBox cbImportant;
30     @FXML
31     private Label lblMessage;
32
33     private final Navigation navigation = new Navigation(); // New instance for Navigation to access the methods to change tabs
34
35     // Method to initialize a certain functionality inside the scene
36     @Override
37     public void initialize(URL url, ResourceBundle resourceBundle) {
38
39         // Lambda expression to prevent users from picking a due date before the current day ( because it does not make sense )
40         dpDueDate.setDayCellFactory(picker -> new DateCell() {
41             public void updateItem(LocalDate date, boolean empty) {
42                 super.updateItem(date, empty);
43                 LocalDate today = LocalDate.now();
44
45                 setDisable(empty || date.compareTo(today) < 0);
46             }
47         });
48     }
49 }
```

Moving on to the TaskCreationController, we can see that I used FXML injection to connect the controls in the FXML file to the controller. All the controllers use this injection. Another thing that all controllers have is the Navigation instance. This is so that users can switch tabs no matter on which scene they are. Going down the code, we can see that I override initialize again. Inside the initialize I have a method **setDayCellFactory()** followed by lambda expressions. This is used to make modifications to the dates in the date picker. In my code, I modified the date picker so that users can only choose the dates starting from the current day to the following days and not backwards, because it would not make sense for a due date to be before the current day.


```

60 // A method which gets called everytime the user presses the "Add" button
61 public void addTask() {
62
63     // Checks if the text field or date picker is empty
64     if (tfTask.getText() == null || tfTask.getText().trim().isEmpty() || dpDueDate.getValue() == null) {
65         lblMessage.setText("Please fill in the Task and Due Date field!"); // If it is, warn the user
66     } else {
67         String taskTitle = tfTask.getText().trim(); // Gets the title of the Task from the text field
68         LocalDate dueDate = dpDueDate.getValue(); // Gets the due date value from the date picker
69         boolean important = cbImportant.isSelected(); // Gets the true or false of whether or not task is important by checking the checkbox
70
71         ToDoItem toDoItem = new ToDoItem(taskTitle, dueDate, important); // Gets the new ToDoItem by using the constructor
72         ToDoData.getInstance().addToDoItem(toDoItem); // Adds the new task to the ToDoItems list
73
74         // Clears all the input fields
75         tfTask.clear();
76         dpDueDate.getEditor().clear();
77         cbImportant.setSelected(false);
78
79         lblMessage.setText("Task successfully added!"); // If everything is filled, displays success message
80
81         // Timer to remove success message after 3.5 seconds
82         new Timeline(new KeyFrame(
83             Duration.millis(3500),
84             event -> lblMessage.setText("")
85         )).play();
86     }
}

```

In the same controller, I have the **addTask()** method. This is called when the “Add” button is clicked in the “Add new task” tab. Inside the method, I have an if statement, checking if the text field for the task title, and the date picker for the due date, has a value. If it does not, there will be a label popping up on the top right warning the user to input all the said fields.

If the user did input all the fields, the controller is going to get the values for the task title, due date and checkbox. Once it gets all the values, it will create a new instance of **ToDoItem**, which will then be added to the *toDoItems* list inside the **ToDoData** class. The controller will then clear the text fields and uncheck the checkbox if it is checked. The program will also display a success message. I also created a new **Timeline** instance to make the success message disappear after 3.5 seconds.

6. Project GitHub Link:

<https://github.com/karsteneugene/ToDoManager>

7. Credits:

- [References for custom classes for ToDoItem and ToDoData from DomUllmann's To-Do List for JavaFX](#)
- [References for custom classes for ToDoItem and ToDoData from bubabi's To-Do List for JavaFX](#)
- [Bro Code's JavaFX Tutorial](#)
- [JavaFX Documentation](#)