To-Do App

From Team Michael

Team members: Michael Awwad

Scenario and Problem Statement

POV: You’re a college student with a packed schedule. You’re juggling:

- Academis: assignments, exams, projects, and deadlines.
- Extracurriculars: club meetings, volunteering, or sports practice
- Personal Life: seeing friends and family, running errands, and personal upkeep tasks.
- Work: part-time shifts that demand time and focus.

Balancing all of this can feel overwhelming, and it’s easy to forget or miss something important. That’s where the To-Do App comes in!
The goal is simple: help users organize their responsibilities effectively. By enabling task prioritization, grouping and management, the app provides a central hub to keep everything in order.

What the System Can Do

01

Add Tasks

Users can create tasks with a name, due date, and priority level (from 1-5)

02

Sort and Group Tasks

Tasks are grouped by their due dates and tasks are sorted by priority

03

Mark Tasks Complete

Once a task is done, users can mark it complete and it's moved to a separate page

04

Manage Completed Tasks

Completed tasks can be restored to the pending tasks list and users can clear all completed tasks with one button

How I Built the System

Task

Represents individual tasks

TaskManager

Handles task logic, such as storing, sorting, and moving tasks between lists

Main

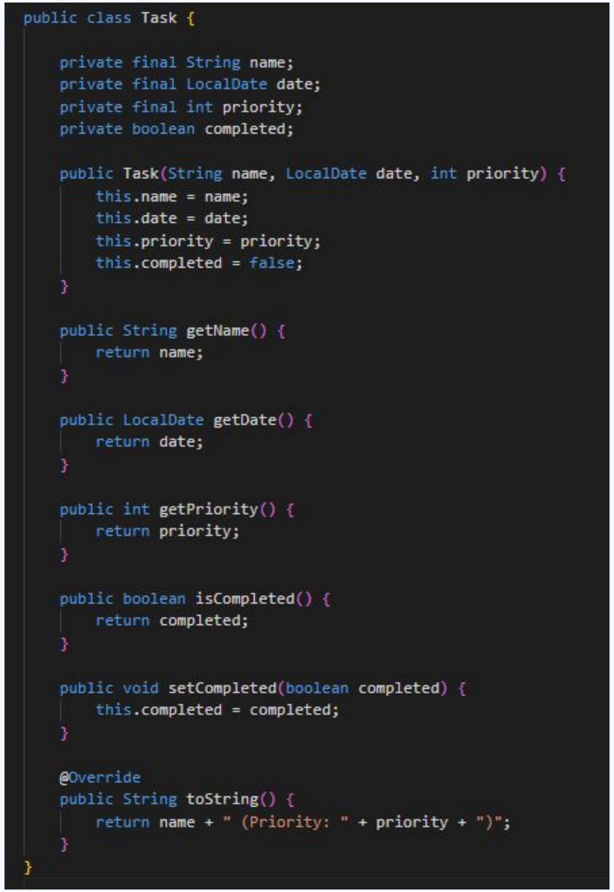
Creates the GUI and handles user interactions

The Task Class

1. Stores task information:
   String name: the tasks title
   LocalDate date: when the task is due
   int priority: how important the task is (1-5)
   boolean isCompleted: whether the task is finished

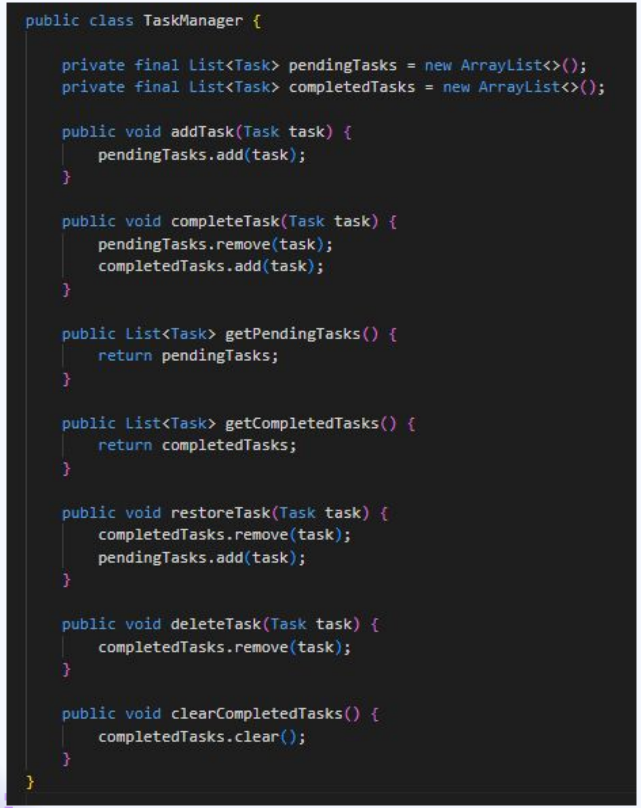
2. Provides methods to interact with tasks, such as **getters, setters,** and a **toString** method to display tasks in the GUI

This class acts as the data model for the app. It was implemented first, as everything else depends on it



```
public class Task {

    private final String name;
    private final LocalDate date;
    private final int priority;
    private boolean completed;

    public Task(String name, LocalDate date, int priority) {
        this.name = name;
        this.date = date;
        this.priority = priority;
        this.completed = false;
    }

    public String getName() {
        return name;
    }

    public LocalDate getDate() {
        return date;
    }

    public int getPriority() {
        return priority;
    }

    public boolean isCompleted() {
        return completed;
    }

    public void setCompleted(boolean completed) {
        this.completed = completed;
    }

    @Override
    public String toString() {
        return name + " (Priority: " + priority + ")";
    }
}
```

The TaskManager Class

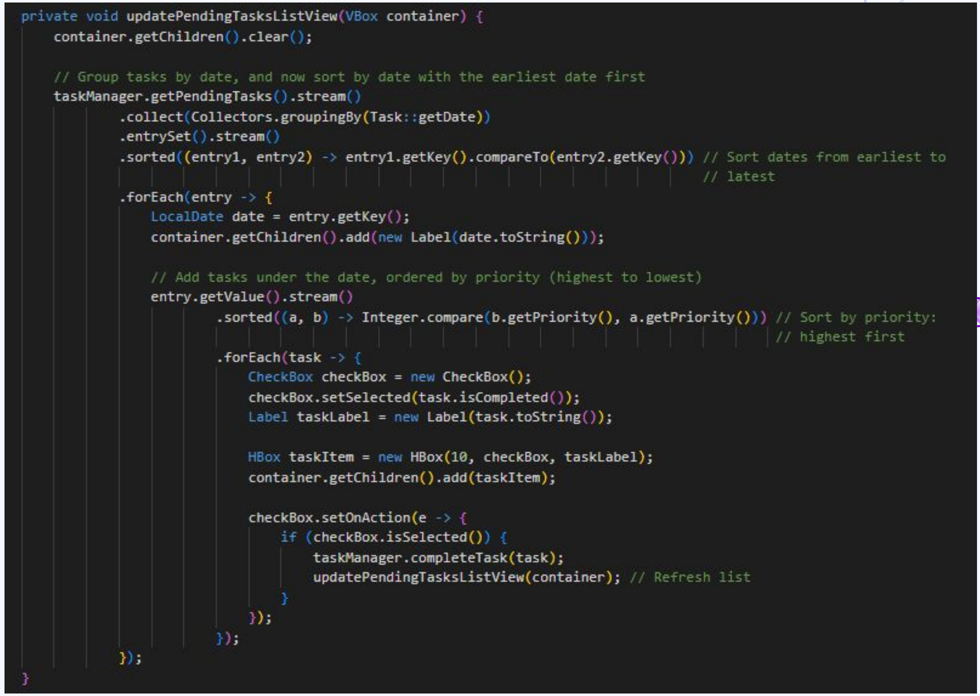


```java
public class TaskManager {

    private final List<Task> pendingTasks = new ArrayList<>();
    private final List<Task> completedTasks = new ArrayList<>();

    public void addTask(Task task) {
        pendingTasks.add(task);
    }

    public void completeTask(Task task) {
        pendingTasks.remove(task);
        completedTasks.add(task);
    }

    public List<Task> getPendingTasks() {
        return pendingTasks;
    }

    public List<Task> getCompletedTasks() {
        return completedTasks;
    }

    public void restoreTask(Task task) {
        completedTasks.remove(task);
        pendingTasks.add(task);
    }

    public void deleteTask(Task task) {
        completedTasks.remove(task);
    }

    public void clearCompletedTasks() {
        completedTasks.clear();
    }
}
```

1. Stores tasks:
   pendingTasks: a list of tasks that are not complete, **completedTasks:** a list of tasks that have been marked complete.
2. The **addTask** method takes a **Task** object and adds it to the **pendingTasks** list.
3. The **completeTask** method moves a task from **pendingTasks** to **completedTasks**, updating its **isCompleted** status.
4. The **deleteTask** method removes a task from the **completedTasks** list.
5. Methods like **getPendingTasks** and **getCompletedTasks** return the current lists for display in the UI.

The Main Class

1. Uses JavaFX to create two scenes: one for pending tasks and another for completed tasks. Each scene is built with **VBox** containers to organize the layout.
2. Buttons for adding tasks, marking them complete, restoring them, and clearing all completed tasks. A **DatePicker** and **Spinner** for users to input due dates and priority.
3. Whenever a task is added, completed, or deleted the UI is refreshed to reflect the change.



```java
private void updatePendingTasksListView(VBox container) {
    container.getChildren().clear();

    // Group tasks by date, and now sort by date with the earliest date first
    taskManager.getPendingTasks().stream()
            .collect(Collectors.groupingBy(Task::getDate))
            .entrySet().stream()
            .sorted((entry1, entry2) -> entry1.getKey().compareTo(entry2.getKey())) // Sort dates from earliest to
                                                                                    // latest
            .forEach(entry -> {
                LocalDate date = entry.getKey();
                container.getChildren().add(new Label(date.toString()));

                // Add tasks under the date, ordered by priority (highest to lowest)
                entry.getValue().stream()
                        .sorted((a, b) -> Integer.compare(b.getPriority(), a.getPriority())) // Sort by priority:
                                                                                             // highest first
                        .forEach(task -> {
                            CheckBox checkBox = new CheckBox();
                            checkBox.setSelected(task.isCompleted());
                            Label taskLabel = new Label(task.toString());

                            HBox taskItem = new HBox(10, checkBox, taskLabel);
                            container.getChildren().add(taskItem);

                            checkBox.setOnAction(e -> {
                                if (checkBox.isSelected()) {
                                    taskManager.completeTask(task);
                                    updatePendingTasksListView(container); // Refresh list
                                }
                            });
                        });
            });
}
```

Here's an example of the method that updates the pending tasks view. This method dynamically organizes tasks by date and priority and updates the UI in real time

Debugging Process

1. Sorting Tasks by Date and Priority:
    - Issue: Task groups were initially sorted from farthest to nearest due date.
    - Solution: I fixed this by explicitly sorting groups in ascending order using **Comparator.comparing**. I also tested edge cases, such as tasks with the same due date but different priorities, to ensure they were displayed correctly.
2. Restore Button:
    - Issue: The restore button didn't update the pending tasks list after restoring a task.
    - Solution: I added a call to **updatePendingTasksListView** immediately after restoring a task, ensuring the UI refreshed automatically.

What I Learned

Debugging

Debugging isn’t just fixing errors, it’s about understanding how and why the errors occur.

Planning

The importance of planning and breaking a project into smaller, manageable parts.

Object Oriented Programming Concepts

How to implement object oriented programming concepts effectively in a real world application.

Live Demo Time