TICKLIST DOCUMENTATION

INTRODUCTION:

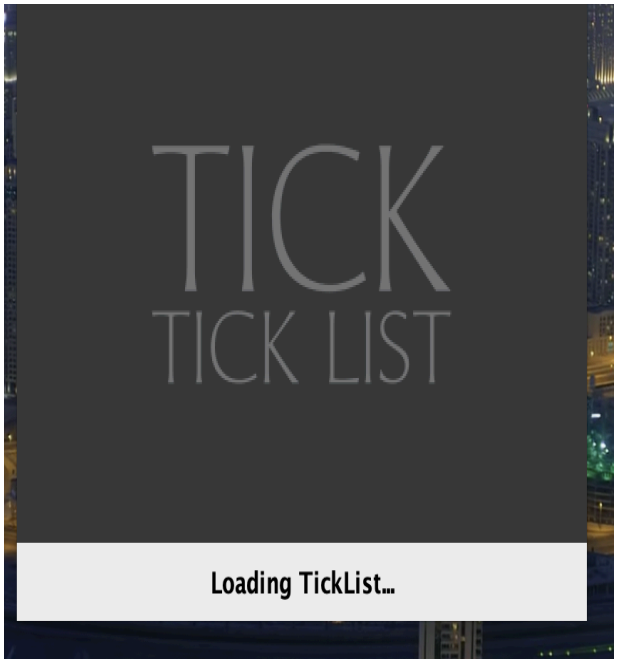
The **TickList Application** is a robust desktop-based to-do list manager designed to streamline task organization and improve productivity. With a focus on user convenience, TickList provides an intuitive interface for creating, managing, and monitoring tasks efficiently. Its integration of visual cues, notifications, and persistence ensures users stay organized and meet deadlines effectively.

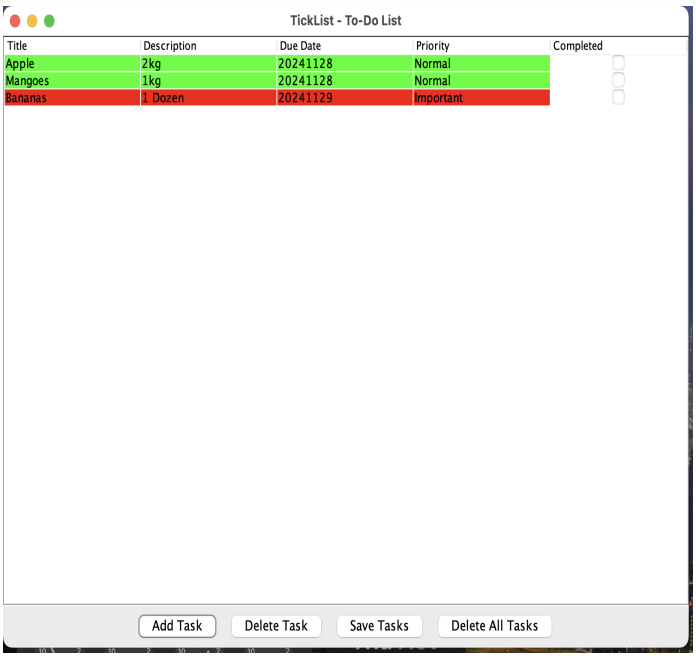
Purpose and Motivation:

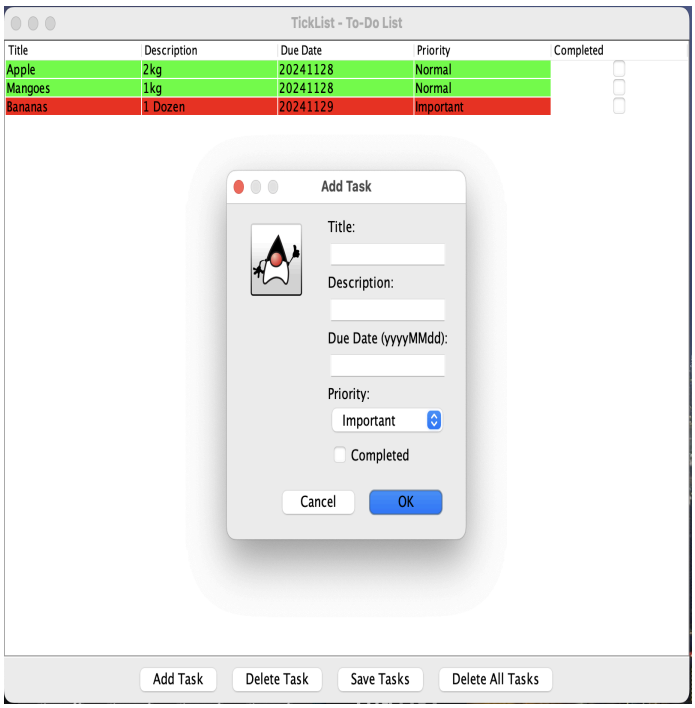
The modern user is often overwhelmed with multiple responsibilities, deadlines, and commitments. Traditional pen-and-paper methods or generic to-do list apps often lack critical features such as reminders, task prioritization, and visual clarity. TickList aims to fill this gap by offering:

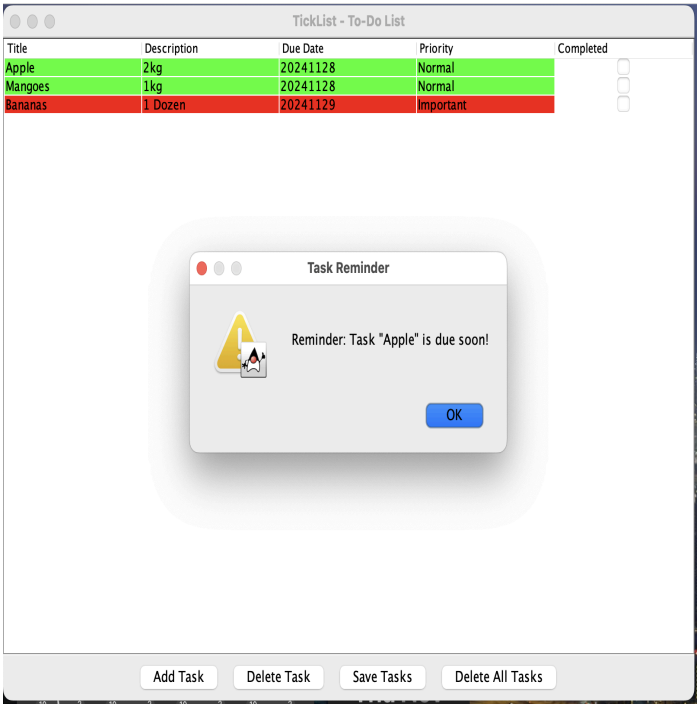
- A clear overview of tasks with **priority-based color coding**.
- Automated reminders for tasks nearing their deadlines.
- **Persistent task storage** to prevent data loss.
- An interactive UI that simplifies task management.

User Interface (UI):









2. Features:

1. **Task Management**:
   - Add tasks with details such as title, description, due date, priority, and completion status.
   - Delete individual tasks or all tasks at once.
   - Mark tasks as completed using a checkbox in the task table.

2. **Visual Enhancements**:
   - Priority-based color coding:
     - **Red**: Important tasks.
     - **Green**: Normal tasks.
     - **Gray**: Completed tasks.
   - Intuitive table-based UI for task visualization.

3. **Reminders and Notifications**:
   - Automatically checks for tasks due within 24 hours.
   - Notifies the user with a pop-up reminder for tasks nearing their deadline.

4. **Persistence**:
    - Tasks are saved to a file (`tasks.ser`) using serialization.
    - Tasks are reloaded on application startup to ensure no data loss.

5. **Splash Screen**:
    - Displays a splash screen with the application logo and loading message during startup.

3. Technical Implementation

3.1 Architecture

- **Programming Language**: Java (Swing for UI)
- **File Structure**:
    - **Task:** Represents individual tasks.
    - **TaskList:** Manages a collection of tasks and handles serialization.
    - **TickListApp:** Main application class for UI and logic.
    - **SplashScreenDemo:** Displays the splash screen at startup.
    - **PriorityTableCellRenderer:** A custom cell renderer for the task table that visually enhances the application by applying color codes to table rows based on the task's priority and completion status.

3.2 Class Descriptions

1. **Task:**
    - Represents a single task with fields for title, description, due date, priority, and completion status.
    - Implements Serializable to enable saving to a file.
    - Provides getters and setters for task attributes.

2. **TaskList:**
    - Manages a list of tasks.
    - Provides methods for adding, removing, and retrieving tasks.
    - Handles serialization (saveTasks) and deserialization (loadTasks) of tasks.

3. **TickListApp:**
    - Main application class responsible for:
        - Task management (add, delete, update tasks).
        - UI setup using JTable and DefaultTableModel.
        - Background reminders for tasks due within 24 hours.

4. **SplashScreenDemo:**
    - Displays a splash screen with the application logo and a loading message before launching the main application.

5. **PriorityTableCellRenderer:** A custom cell renderer for the task table that visually enhances the application by applying color codes to table rows based on the task's priority and completion status.

   Functionality:

   - **Red**: Tasks marked as "Important".
   - **Green**: Tasks marked as "Normal".
   - **Gray**: Tasks that are completed.

     Ensures that selected rows retain their highlighting for better usability.

3.3 Reminder Logic

- A Timer is used to periodically check for tasks due within 24 hours.
- Steps:
    - Parse the task’s due date using SimpleDateFormat.
    - Compare the due date with the current time.
    - If a task is due within 24 hours, display a pop-up notification.

4. Usage Instructions

1. **Adding a Task**:

- Click the "Add Task" button.
- Fill in the details (title, description, due date in YYYYMMDD format, priority).
- Save the task to see it added to the task list.

2. **Marking a Task as Completed**:
   - Check the checkbox in the **"Completed"** column of the table.
   - The row will turn gray to indicate completion.

3. **Deleting Tasks**:
   - Select a task and click **"Delete Task"** to remove it.
   - Click **"Delete All Tasks"** to clear all tasks.

4. **Receiving Reminders**:
   - The application automatically checks for tasks due within 24 hours.
   - A pop-up notification will appear for tasks nearing their deadline.

5. Key Code Highlights:

5.1 Reminder Logic:

```
        startReminderChecker();

        // Show the frame
        frame.setVisible(true);
    }

    private void startReminderChecker() {  1 usage
        // Run the reminder checker every 1 minute (60,000 milliseconds)
        Timer timer = new Timer( delay: 60000, e -> checkForDueTasks());
        timer.start();
    }
```

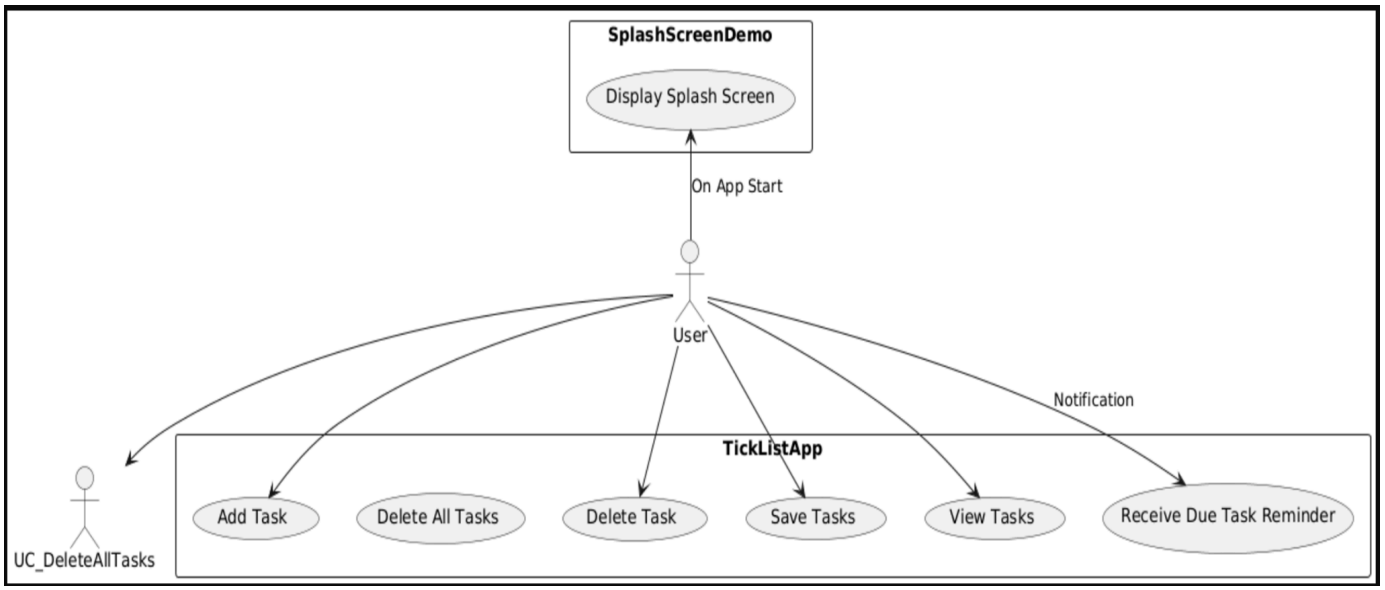
```java
private void checkForDueTasks() {  1 usage
    Date now = new Date();
    SimpleDateFormat sdf = new SimpleDateFormat( pattern: "yyyyMMdd"); // Assuming due date is in YYYYMMDD format

    for (Task task : taskList.getTasks()) {
        try {
            Date dueDate = sdf.parse(task.getDueDate());

            dueDate = new Date(dueDate.getTime() + (1000 * 60 * 60 * 24 - 1));

            // Check if the task is due in the next 24 hours
            long timeDifference = dueDate.getTime() - now.getTime();
            long hoursDifference = timeDifference / (1000 * 60 * 60);

            if (hoursDifference <= 24 && hoursDifference >= 0 && !task.isCompleted()) {
                // Notify the user
                JOptionPane.showMessageDialog(frame,  message: "Reminder: Task \"" + task.getTitle() + "\" is due soon!",  title: "Task Reminder", JOptionPane.WARNING_MESSAGE);
            }
        } catch (ParseException e) {
            System.err.println("Invalid date format for task: " + task.getTitle());
        }
    }
}
```

5.2 Priority-Based Row Coloring:

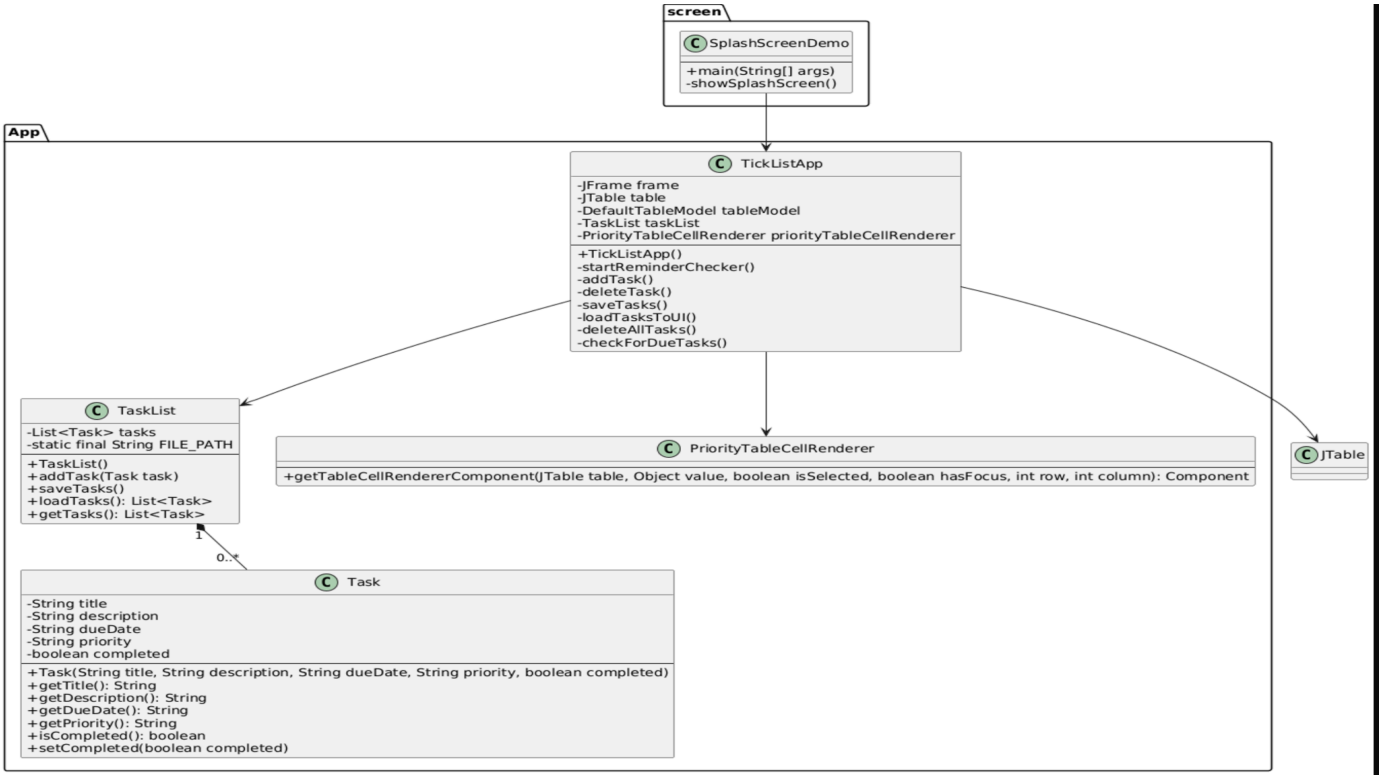
TickListApp.java | PriorityTableCellRenderer.java | SplashScreenDemo.java | Task.java | TaskList.java

```java
package App;

import javax.swing.*;
import javax.swing.table.DefaultTableCellRenderer;
import java.awt.*;

class PriorityTableCellRenderer extends DefaultTableCellRenderer {  1 usage
    @Override
    public Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected, boolean hasFocus, int row, int column) {
        // Calling the parent method to get the default renderer
        Component cell = super.getTableCellRendererComponent(table, value, isSelected, hasFocus, row, column);

        String priority = (String) table.getValueAt(row,  column: 3);  // Priority is in the 4th column (index 3)
        Boolean completed = (Boolean) table.getValueAt(row,  column: 4);  // Completed is in the 5th column (index 4)

        if (Boolean.TRUE.equals(completed)) {
            cell.setBackground(Color.LIGHT_GRAY); // Gray for completed tasks
        } else if ("Important".equals(priority)) {
            cell.setBackground(Color.RED); // Red for important tasks
        } else if ("Normal".equals(priority)) {
            cell.setBackground(Color.GREEN); // Green for normal tasks
        } else {
            cell.setBackground(Color.WHITE); // Default background
        }

        // Retain the selection highlighting if the row is selected
        if (isSelected) {
            cell.setBackground(table.getSelectionBackground());
            cell.setForeground(table.getSelectionForeground());
        } else {
            cell.setForeground(Color.BLACK); //default
        }

        return cell;
    }
}
```

6. Use-Case Diagram:



7. Class Diagram:



8. Testing and Validation

Test Case: PriorityTableCellRendererDemoTest

- **Description**: Tests the background color of table cells based on task priority and completion status.
- **Test Method**: `testImportantTaskNotCompleted()`
- **Expected Outcome**: Background color of "Important" tasks should be red when not completed.

Test Case: SplashScreenDemoTest

- **Description:** Verifies that the splash screen logo loads correctly and maintains the correct aspect ratio and positioning.
- **Test Method:** `testLogoResourceLoading()` and many more…
- **Expected Outcome:** Logo should load successfully, and its image load status should be complete.

Test Case: TaskManagerTestDemo

- **Description:** Tests the core functionalities of adding, deleting, and editing tasks in the TaskManager class.
- **Test Method**: `testAddTask()`
- **Expected Outcome:** After adding a task, the task list size should be 1, and the task should match the added one.

Test Case: TickListTestDemo

- **Description:** Verifies the task management in TaskList, including adding, editing, deleting tasks, and checking for reminders for tasks due soon.
- **Test Method:** `testReminderForDueTasks()`
- **Expected Outcome:** Only tasks due within 24 hours and not completed should be counted as due soon.

9. Conclusion:

The TickList Application successfully combines essential task management features with visual and functional enhancements. The addition of reminders ensures users stay on top of deadlines, making the application a useful tool for productivity.