Student Name: [Your Name]

Date: July 27, 2025

Java Final Project Journal – To‑Do List Manager

Part 1: Defining Your Problem

Many people keep track of errands, assignments and reminders on scraps of paper or in their heads. Without a structured system, tasks can easily be forgotten or completed late. The goal of this project is to design and implement a simple to‑do list manager that runs in the command line. The program will allow a user to create tasks with a short description, a due date (in the format YYYY‑MM‑DD), and a priority level. Users can view all tasks, mark tasks as completed, remove tasks, and save or load their list to or from a file. The program solves the problem of task organization by keeping all necessary information in one place and presenting it in an ordered, easy‑to‑read format.

Expected input data:

• A task description (text describing what needs to be done).

• A due date in the form of a calendar date (YYYY‑MM‑DD).

• A priority value (a positive integer where 1 represents highest priority).

Program behaviour:

• When adding a task, the program will validate the due date and priority to ensure they are in the correct format.

• The user can repeatedly add tasks, view them sorted by priority and due date, mark tasks as completed, remove them, and save the list.

• Tasks are stored in memory while the program runs and can be saved to a text file so they persist between runs.

Expected outputs/results:

• Confirmation messages when tasks are added, marked as completed, removed, or saved.

• A neatly formatted list of tasks showing the description, due date, priority and completion status.

• A persistent file (tasks.txt) containing all tasks so that they can be reloaded in a later session.

Part 2: Working Through Specific Examples

To better understand how the program should work, consider the following example scenario:

1. The user starts the program. No tasks exist initially, so the list is empty.

2. The user selects the option to add a new task. They enter "Finish reading chapter 5 of the Java textbook" as the description, enter a due date of 2025‑08‑01 and choose priority 1 (highest priority). The program validates the date and priority, adds the task to the list and confirms the addition.

3. The user adds a second task with the description "Buy groceries" due on 2025‑08‑02 and priority 2. Again, the program confirms the task has been added.

4. When the user chooses to view all tasks, the program displays both tasks sorted by priority and due date. Each task is numbered and shows its description, due date, priority and status (Pending).

5. The user marks the second task as completed by entering the task number. The program updates the task’s status to Completed and confirms the change.

6. The user chooses to save tasks. The program writes both tasks to a file named tasks.txt so they can be reloaded later.

7. Exiting and restarting the program, the tasks are automatically loaded from tasks.txt and displayed again when the user views the list. Completed tasks remain on the list but are clearly marked as completed.

This example demonstrates the basic flow of interaction with the program, including adding tasks, viewing them, marking a task as completed, and saving tasks so they persist between sessions. The specific results match the expected program behaviour described in Part 1.

Part 3: Generalizing Into Pseudocode

The following pseudocode outlines the general logic of the to‑do list manager. It describes how the program should behave regardless of specific input values and refers to Java constructs where appropriate:

• Start program.

• Initialize an empty list to store tasks.

• Load tasks from "tasks.txt" if it exists.

• Repeat until the user chooses to exit:

– Display a menu of options: Add a task, View tasks, Mark a task as completed, Remove a task, Save tasks, Exit.

– Read the user’s choice.

– If the choice is "Add":

• Prompt for task description (String).

• Prompt for due date in YYYY‑MM‑DD format and validate it (use LocalDate.parse).

• Prompt for priority as a positive integer and validate it (use Integer.parseInt).

• Create a new Task object with the provided data, set its completed flag to false, and add it to the list.

• Display a confirmation message.

– Else if the choice is "View":

• If the list is empty, inform the user.

• Otherwise, make a copy of the list, sort it by priority (ascending) and due date (earliest first).

• For each task in the sorted list, display its number and details: description, formatted due date, priority, and status (Pending or Completed).

– Else if the choice is "Mark as completed":

• If the list is empty, inform the user.

• Otherwise, display the tasks with numbers.

• Prompt for the task number. Validate that it is within range.

• If the selected task is already completed, inform the user. Otherwise, set its completed flag to true and confirm.

– Else if the choice is "Remove":

• If the list is empty, inform the user.

• Otherwise, display the tasks with numbers.

• Prompt for the task number. Validate the input.

• Remove the selected task from the list and confirm its removal.

– Else if the choice is "Save":

• Open (or create) "tasks.txt" for writing.

• For each task in the list, write a line containing the task’s description, due date (YYYY‑MM‑DD), priority and completion status separated by the pipe character.

• Close the file and inform the user that the tasks have been saved.

– Else if the choice is "Exit":

• Save tasks automatically so the user’s data is not lost.

• Display a goodbye message and terminate the loop.

– Otherwise: notify the user of an invalid option.

• End program.

Part 4: Testing Your Program

Testing was carried out by running the program and exercising each of its features. Below are some representative test cases, the expected outcome, and any issues encountered during development:

• Test Case 1 – Adding and viewing tasks:

– Input: Add two tasks with different priorities and due dates as in the example above.

– Expected outcome: The program confirms each addition and, when viewing the list, displays both tasks sorted by priority and due date.

– Result: The tasks were displayed correctly. No errors occurred.

• Test Case 2 – Marking a task as completed:

– Input: After adding tasks, choose to mark the second task as completed and input its number.

– Expected outcome: The task’s status changes to Completed and is reflected in the next view.

– Result: Initially an IndexOutOfBoundsException occurred when entering an invalid task number. The code was corrected to validate the user’s input and prompt again until a valid number is provided.

• Test Case 3 – Removing a task:

– Input: Add several tasks, then choose to remove the first task.

– Expected outcome: The task is removed from the list and does not appear the next time the list is viewed.

– Result: The program performed as expected after adding input validation to ensure the task number was within range.

• Test Case 4 – Saving and loading tasks:

– Input: After adding and modifying tasks, save them, exit the program, restart it, and load the tasks automatically.

– Expected outcome: The program should recreate the previous state of the task list, including completion status.

– Result: Tasks were correctly written to and read from tasks.txt. A NumberFormatException occurred when reading a malformed line; a guard was added to skip lines that cannot be parsed.

These tests helped verify that user input is properly validated and that the program can handle common errors gracefully. Whenever an exception was thrown, I examined the stack trace in my IDE, identified the cause (such as parsing invalid input or using an index out of range) and updated the code to handle the situation by asking the user to retry or by skipping bad data.

Part 5: Commenting Your Program

Below is the complete Java source code for the to‑do list manager. Comments throughout the code explain the purpose of each class, method and major block of logic in plain English so that someone unfamiliar with Java can understand how the program works.

import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.File;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

import java.time.LocalDate;

import java.time.format.DateTimeFormatter;

import java.time.format.DateTimeParseException;

import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;

/\*\*

\* A simple command‑line based to‑do list manager. The program allows

\* the user to add tasks, view the current list of tasks, mark a task

\* as completed, remove a task, save the list to a file and load

\* existing tasks from a file. Tasks include a description, due date,

\* priority level and completion status. This class contains the

\* entry point (main method) and manages the menu loop and user

\* interaction.

\*/

public class ToDoListManager {

/\*\*

\* Inner class representing a single to‑do list task. Each task

\* stores a short description, a due date, a priority level (1 =

\* highest priority) and a flag indicating whether it has been

\* completed. Providing getters and setters makes it easier to

\* update these values later if needed. The toString method is

\* overridden to provide a human readable representation of a task

\* when printing the list to the console.

\*/

private static class Task {

private String description;

private LocalDate dueDate;

private int priority;

private boolean completed;

public Task(String description, LocalDate dueDate, int priority) {

this.description = description;

this.dueDate = dueDate;

this.priority = priority;

this.completed = false;

}

public String getDescription() {

return description;

}

public LocalDate getDueDate() {

return dueDate;

}

public int getPriority() {

return priority;

}

public boolean isCompleted() {

return completed;

}

public void setCompleted(boolean completed) {

this.completed = completed;

}

@Override

public String toString() {

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy‑MM‑dd");

String status = completed ? "Completed" : "Pending";

return String.format(

"Description: %s\nDue Date: %s\nPriority: %d\nStatus: %s",

description,

dueDate.format(formatter),

priority,

status);

}

}

// List to store the tasks in memory

private List<Task> tasks;

// Scanner to read user input from standard input

private Scanner scanner;

// Formatter used to parse and format dates in a consistent way

private static final DateTimeFormatter DATE\_FORMAT = DateTimeFormatter.ofPattern("yyyy‑MM‑dd");

// Name of the file where tasks are saved. Using a plain text file

// with pipe‑separated values keeps the storage simple and human

// readable.

private static final String FILE\_NAME = "tasks.txt";

/\*\*

\* Constructs a new to‑do list manager. This constructor

\* initializes the tasks list, the scanner and loads any existing

\* tasks from disk. Loading existing tasks ensures that users

\* don't lose their list between program runs. If the file is

\* missing or cannot be read, the program will simply start with

\* an empty list.

\*/

public ToDoListManager() {

tasks = new ArrayList<>();

scanner = new Scanner(System.in);

loadTasks();

}

/\*\*

\* Entry point. Creates an instance of the manager and starts

\* interacting with the user via a menu. The program will

\* continue to run until the user chooses to exit.

\*

\* @param args command line arguments are ignored for this program

\*/

public static void main(String[] args) {

ToDoListManager manager = new ToDoListManager();

manager.run();

}

/\*\*

\* Main menu loop. The loop presents a set of options to the user

\* and dispatches to the appropriate handler based on the user

\* selection. Menu options include adding a task, viewing tasks,

\* marking tasks as completed, removing tasks, saving tasks to a

\* file and exiting the program. Input is validated so that

\* invalid selections don't cause the program to crash. The loop

\* continues until the user selects the exit option.

\*/

private void run() {

boolean running = true;

while (running) {

System.out.println();

System.out.println("=== To‑Do List Manager ===");

System.out.println("1. Add a new task");

System.out.println("2. View all tasks");

System.out.println("3. Mark a task as completed");

System.out.println("4. Remove a task");

System.out.println("5. Save tasks to file");

System.out.println("6. Exit");

System.out.print("Enter your choice: ");

String choice = scanner.nextLine();

switch (choice) {

case "1":

addTask();

break;

case "2":

viewTasks();

break;

case "3":

markTaskCompleted();

break;

case "4":

removeTask();

break;

case "5":

saveTasks();

break;

case "6":

// Save tasks automatically before exiting to avoid data loss

saveTasks();

System.out.println("Goodbye!");

running = false;

break;

default:

System.out.println("Invalid choice. Please select a valid option.");

}

}

}

/\*\*

\* Prompts the user to enter details for a new task and adds it to

\* the in‑memory list. The method asks for a description, a due

\* date in the format yyyy‑MM‑dd and a priority level. Input

\* validation ensures that the user enters a valid date and

\* priority. If invalid input is entered, the user is prompted

\* again until a valid value is provided.

\*/

private void addTask() {

System.out.print("Enter task description: ");

String description = scanner.nextLine().trim();

// Validate date input. Use a loop to keep asking until a valid

// date is entered. If the date string cannot be parsed, an

// exception is caught and the user is notified.

LocalDate dueDate = null;

while (dueDate == null) {

System.out.print("Enter due date (YYYY‑MM‑DD): ");

String dateInput = scanner.nextLine().trim();

try {

dueDate = LocalDate.parse(dateInput, DATE\_FORMAT);

} catch (DateTimeParseException e) {

System.out.println("Invalid date format. Please use YYYY‑MM‑DD.");

}

}

// Validate priority input. Priority must be a positive integer. A

// similar loop is used to ensure the user provides a valid

// number.

int priority = 0;

while (priority <= 0) {

System.out.print("Enter priority (1 = highest priority): ");

String priorityInput = scanner.nextLine().trim();

try {

priority = Integer.parseInt(priorityInput);

if (priority <= 0) {

System.out.println("Priority must be a positive integer.");

}

} catch (NumberFormatException e) {

System.out.println("Invalid number. Please enter a valid integer for priority.");

}

}

Task newTask = new Task(description, dueDate, priority);

tasks.add(newTask);

System.out.println("Task added successfully!");

}

/\*\*

\* Displays the current list of tasks. Each task is printed with

\* its index in the list so that the user can refer to it when

\* marking it completed or removing it. The list is sorted by

\* priority (highest priority first) and then by due date so that

\* urgent tasks appear at the top. If no tasks exist, the user

\* is informed accordingly.

\*/

private void viewTasks() {

if (tasks.isEmpty()) {

System.out.println("No tasks found.");

return;

}

// Create a copy of the task list so that sorting does not

// modify the order in the original list. The copy is sorted

// by priority (ascending priority number means higher priority)

// and then by due date. Completed tasks remain in the list

// but are flagged as such.

List<Task> sortedTasks = new ArrayList<>(tasks);

sortedTasks.sort((a, b) -> {

if (a.getPriority() != b.getPriority()) {

return Integer.compare(a.getPriority(), b.getPriority());

}

return a.getDueDate().compareTo(b.getDueDate());

});

System.out.println("Current tasks:");

int index = 1;

for (Task task : sortedTasks) {

System.out.println("--- Task #" + index + " ---");

System.out.println(task);

index++;

}

}

/\*\*

\* Allows the user to mark a task as completed. The user is

\* prompted for the index of the task. Input is validated to

\* ensure that a valid index is provided. Completed tasks remain

\* in the list but are flagged as completed so they can be

\* differentiated from pending tasks when viewing the list.

\*/

private void markTaskCompleted() {

if (tasks.isEmpty()) {

System.out.println("No tasks to mark as completed.");

return;

}

// Display tasks so that the user knows the indices

viewTasks();

int index = -1;

while (index < 1 || index > tasks.size()) {

System.out.print("Enter the task number to mark as completed: ");

String input = scanner.nextLine().trim();

try {

index = Integer.parseInt(input);

if (index < 1 || index > tasks.size()) {

System.out.println("Invalid task number. Please try again.");

}

} catch (NumberFormatException e) {

System.out.println("Please enter a valid task number.");

}

}

Task task = tasks.get(index - 1);

if (task.isCompleted()) {

System.out.println("Task is already marked as completed.");

} else {

task.setCompleted(true);

System.out.println("Task marked as completed!");

}

}

/\*\*

\* Removes a task from the list. The user is prompted for the

\* index of the task to remove. Input is validated to ensure that

\* a valid index is provided and the removal is carried out

\* accordingly. If the user enters an invalid index, they are

\* prompted again until a valid one is provided.

\*/

private void removeTask() {

if (tasks.isEmpty()) {

System.out.println("No tasks to remove.");

return;

}

viewTasks();

int index = -1;

while (index < 1 || index > tasks.size()) {

System.out.print("Enter the task number to remove: ");

String input = scanner.nextLine().trim();

try {

index = Integer.parseInt(input);

if (index < 1 || index > tasks.size()) {

System.out.println("Invalid task number. Please try again.");

}

} catch (NumberFormatException e) {

System.out.println("Please enter a valid task number.");

}

}

Task removed = tasks.remove(index - 1);

System.out.println("Removed task: " + removed.getDescription());

}

/\*\*

\* Loads tasks from the persistent storage file. Each line in the

\* file represents one task and uses the format

\* description|dueDate|priority|completed. If the file is not

\* found or cannot be read, this method quietly returns without

\* affecting the current list of tasks. If the file is present,

\* tasks are cleared before loading to avoid duplicating tasks.

\*/

private void loadTasks() {

File file = new File(FILE\_NAME);

if (!file.exists()) {

return;

}

try (BufferedReader reader = new BufferedReader(new FileReader(file))) {

String line;

tasks.clear();

while ((line = reader.readLine()) != null) {

// Skip empty lines

if (line.trim().isEmpty()) {

continue;

}

String[] parts = line.split("\\|");

if (parts.length != 4) {

continue; // Malformed line; skip it

}

String description = parts[0];

LocalDate date;

try {

date = LocalDate.parse(parts[1], DATE\_FORMAT);

} catch (DateTimeParseException e) {

continue; // Skip invalid date entries

}

int pr;

try {

pr = Integer.parseInt(parts[2]);

} catch (NumberFormatException e) {

pr = 1; // Default priority

}

boolean comp = Boolean.parseBoolean(parts[3]);

Task task = new Task(description, date, pr);

task.setCompleted(comp);

tasks.add(task);

}

} catch (IOException e) {

System.err.println("Error reading tasks from file: " + e.getMessage());

}

}

/\*\*

\* Saves the current list of tasks to a file. Each task is

\* written on its own line in the format

\* description|dueDate|priority|completed. If an error occurs

\* during writing (for example, if the file cannot be created), an

\* error message is printed to the console. This method is

\* called when the user chooses to save or when exiting the

\* program.

\*/

private void saveTasks() {

try (BufferedWriter writer = new BufferedWriter(new FileWriter(FILE\_NAME))) {

for (Task task : tasks) {

String line = String.join("|", new String[] {

task.getDescription(),

task.getDueDate().format(DATE\_FORMAT),

Integer.toString(task.getPriority()),

Boolean.toString(task.isCompleted())

});

writer.write(line);

writer.newLine();

}

System.out.println("Tasks saved successfully to " + FILE\_NAME);

} catch (IOException e) {

System.err.println("Error saving tasks: " + e.getMessage());

}

}

}

Part 6: Your Completed Program

To ensure that others can view and run your program, create a share link from your IDE or code hosting service (for example, repl.it or an online Java IDE). Paste the link here so that your instructor can access your full program. If your IDE does not support sharing, be sure to provide enough detail in Part 5 so that someone can compile and run your code.

IDE Share Link: <insert your link here>