```
import java.io.FileNotFoundException;

/**
 * This main class reads in the tasks from the file, waits for
 * user input, and responds to the command issued by the user
 * appropriately.
 * @author Tanner
 *
 */
public class KrewsonToodle {

    /**
     * The main method of this class executes when the program
     * is first launched, and fulfills the intended purpose of
     * the program.
     * @param args
     */
    public static void main(String[] args) {

        Console c = new Console();
        ToodleFile tf = new ToodleFile();
        TaskList tl = new TaskList();

        try {
            tl = new TaskList(tf.readTasksFromFile());
        } catch (IOException e) {
            c.printLine(e.getMessage());
            System.exit(0);
        }

        while (true) {

            //validate the input
            int cmd = 0;
            boolean inputValid = false;
            while (!inputValid) {
                try {
                    cmd = Integer.parseInt(c.promptUserFor("Please enter a 
command (1 - 7): "));
                    inputValid = true;
                } catch (Exception e) {}

            }

            //respond to the input
            switch (cmd) {
            case 1:
                String description = c.promptUserFor("Enter task description: 
");
                char priority = c.promptUserFor("Enter priority (U = urgent, N = 
normal priority, L = low priority): ").toUpperCase().charAt(0);
                int taskOrder = Integer.parseInt(c.promptUserFor("Enter task 
order: "));
                int idOfNewTask = tl.createNewTask(description, priority, 
taskOrder);
                c.printLine("-- Task Created: ID is " + idOfNewTask);
                break;
            case 2:
                c.printTasks(tl.getAllIncompleteTasks());
                break;
            case 3:
                priority = c.promptUserFor("Please enter a priority (U, N, L): 
").toUpperCase().charAt(0);
                c.printTasks(tl.getAllIncompleteTasksWithPriority(priority));
                break;
            case 4:
                c.printTasksWithStatus(tl.getAllTasksInOrder());
                break;
            case 5:
                int idToComplete = Integer.parseInt(c.promptUserFor("Enter ID of 
task to mark as completed: "));
                tl.completeTask(idToComplete);
                c.printLine("-- Task " + idToComplete + " marked as completed 
--");
                break;
            case 6:
                int idToCancel = Integer.parseInt(c.promptUserFor("Enter ID of 
task to cancel: "));
                String reason = c.promptUserFor("Enter reason for cancellation: 
");

                tl.cancelTask(idToCancel, reason);
                c.printLine("-- Task " + idToCancel + " cancelled --");
                break;
            case 7:
                c.printLine("Goodbye");
                try {
                    tf.writeTasksToFile(tl.getAllTasksInOrder());
                } catch (FileNotFoundException e) {
                    c.printLine("Failed to save task list to Task_List.txt");
                    System.exit(0);
                }
                System.exit(0);
            default:
                break;
            }
            c.printLine("");
        }

    }

}
```

```java
/**
package edu.truman.cs260.krewson.toodle;

/**
 * Represents a task that has been identified as
 * cancelled by the user.
 * @author Tanner
 *
 */
public class CancelledTask extends Task {

    /**
     * Stores the reason the user cancelled the task.
     */
    private String cancellationReason;

    /**
     * This constructor takes in all of the necessary
     * information to create a task, plus the information
     * specific to a cancelled task.
     * @param identifier unique internal id
     * @param description name of the task
     * @param priority urgent, normal, or low priority
     * @param order priority within the three priority levels
     * @param cancellationReason the reason the user cancelled the task
     */
    public CancelledTask(int identifier, String description, char priority, int
order, String cancellationReason) {
        super(identifier, description, priority, order, "Cancelled");
        this.cancellationReason = cancellationReason;
    }

    /**
     * Get the reason the user cancelled this task.
     * @return the reason the user cancelled the task
     */
    public String getCancellationReason () {
        return this.cancellationReason;
    }

}
```

```
/**
package edu.truman.cs260.krewson.toodle;

import java.util.Date;

/**
 * Represents a task that has been identified as
 * cancelled by the user.
 * @author Tanner
 *
 */
public class CompletedTask extends Task {

    /**
     * Stores the date the user completed the task.
     */
    private Date completionDate;

    /**
     * This constructor takes in all of the necessary
     * information to create a task, plus the information
     * specific to a completed task.
     * @param identifier unique internal id
     * @param description name of the task
     * @param priority urgent, normal, or low priority
     * @param order priority within the three priority levels
     * @param completionDate the date the user completed the task
     */
    public CompletedTask(int identifier, String description, char priority, int
order, Date completionDate) {
        super(identifier, description, priority, order, "Completed");
        this.completionDate = completionDate;
    }

    /**
     * Get the date the user completed this task.
     * @return the date the user completed the task
     */
    public Date getCompletionDate () {
        return this.completionDate;
    }

}
```

```java
/**
package edu.truman.cs260.krewson.toodle;

import java.io.PrintStream;

/**
 * Wraps the regular Java console and provides
 * functionality specific to printing Tasks.
 * @author Tanner
 *
 */
public class Console {

    /**
     * Wraps the user input from the console.
     */
    private Scanner consoleIn = new Scanner(System.in);

    /**
     * Wraps the regular output to the console.
     */
    private PrintStream consoleOut = System.out;

    /**
     * Asks the user a question, and returns the
     * response.
     * @param query the question to ask the user
     * @return the value the user enters
     */
    public String promptUserFor (String query) {
        consoleOut.print(query);
        return consoleIn.nextLine();
    }

    /**
     * Prints a list of the tasks in a given
     * ArrayList of tasks in the format specified
     * in the functional spec.
     * @param tasksToPrint the tasks to print
     */
    public void printTasks (ArrayList<Task> tasksToPrint) {
        consoleOut.println("ID     Task                 Priority Order");
        consoleOut.println("------ ----                 -------- -----");

        for (Task task : tasksToPrint) {
            consoleOut.printf("%1$6d %2$-20.20s %3$8.8s %4$5d",
                    task.getIdentifier(),
                    task.getDescription(),
                task.getPriority(),
                task.getOrder());
            consoleOut.println();
        }
    }

    /**
     * Prints a list of the tasks in a given
     * ArrayList of tasks in the format specified
     * in the functional spec and includes the additional
     * information found in CompletedTasks and
     * CancelledTasks.
     * @param tasksToPrint the tasks to print
     */
    public void printTasksWithStatus (ArrayList<Task> tasksToPrint) {
        consoleOut.println("ID     Task                 Priority Order Status
Date     Reason");
        consoleOut.println("------ ----                 -------- ----- ------
-------- ------");

        for (Task task : tasksToPrint) {
            consoleOut.printf("%1$6d %2$-20.20s %3$8.8s %4$5d ",
                    task.getIdentifier(),
                    task.getDescription(),
                    task.getPriority(),
                    task.getOrder());
            if (task instanceof CompletedTask) {
                CompletedTask thisTask = (CompletedTask)task;
                consoleOut.printf("%1$-6.6s%2$9tD ",
                        thisTask.STATUS,
                        thisTask.getCompletionDate());
            } else if (task instanceof CancelledTask) {
                CancelledTask thisTask = (CancelledTask)task;
                consoleOut.printf("%1$-6.6s          %2$-1s",
                        thisTask.STATUS,
                        thisTask.getCancellationReason());
            }
            consoleOut.println();
        }
    }

    /**
     * Wraps the regular System.out.println
     * @param lineToPrint the line to print
     */
    public void printLine (String lineToPrint) {
        consoleOut.println(lineToPrint);
    }
```

```
}
```

```
/**
package edu.truman.cs260.krewson.toodle;

/**
 * Represents a task that has not yet been completed
 * by the user.
 * @author Tanner
 *
 */
public class IncompleteTask extends Task {

    /**
     * This constructor takes in all of the necessary
     * information to create a task that has yet to
     * be completed.
     * @param identifier unique internal id
     * @param description name of the task
     * @param priority urgent, normal, or low priority
     * @param order priority within the three priority levels
     */
    public IncompleteTask(int identifier, String description, char priority, int
order) {
        super(identifier, description, priority, order, "Incomplete");
    }

}
```

```java
/**
package edu.truman.cs260.krewson.toodle;

/**
 * Represents a task that would find itself on a
 * to-do list.
 * @author Tanner
 *
 */
public class Task {

    /**
     * Stores the information relating to the task,
     * its id, description, priority, and order within
     * its priority.
     */
    private int identifier;
    private String description;
    private char priority;
    private int order;

    /**
     * Stores the status of the type of Task.
     * This is to be assigned by any extending class.
     */
    public final String STATUS;

    /**
     * This constructor takes in all of the necessary
     * information to create a task.
     * @param identifier unique internal id
     * @param description name of the task
     * @param priority urgent, normal, or low priority
     * @param order priority within the three priority levels
     * @param STATUS the type of task
     */
    public Task(int identifier, String description, char priority, int order,
String STATUS) {
        this.identifier = identifier;
        this.description = description;
        this.priority = priority;
        this.order = order;

        this.STATUS = STATUS;
    }

    /**
     * Gets the task's identifier.
     * @return unique internal id
     */
    public int getIdentifier () {
        return this.identifier;
    }

    /**
     * Gets the task's description.
     * @return name of the task
     */
    public String getDescription () {
        return this.description;
    }

    /**
     * Gets the task's priority.
     * @return urgent, normal, or low priority
     */
    public char getPriority () {
        return this.priority;
    }

    /**
     * Gets the task's order.
     * @return priority within the three priority levels
     */
    public int getOrder () {
        return this.order;
    }

    /**
     * Compares this task to another task by its
     * priority order. Returns positive if it is of
     * higher priority, negative if it is of lower
     * priority, and 0 if they are of the same
     * priority.
     * @param other the task to compare this one to
     * @return a positive or negative number, or 0
     */
    public int compareToByOrder (Task other) {
        if (this.getPriorityStrength() > other.getPriorityStrength()) {
            return 1;
        } else if (this.getPriorityStrength() < other.getPriorityStrength()) {
            return -1;
        } else {
            if (this.getOrder() > other.getOrder()) {
                return 1;
            } else if (this.getOrder() < other.getOrder()) {
                return -1;
            } else {
                return 0;
            }
        }
    }

    /**
     * Compares this task to another task by its
     * identifier. Returns positive if it is of
     * higher priority, negative if it is of lower
     * priority, and 0 if they are of the same
     * priority.
     * @param other the task to compare this one to
     * @return a positive or negative number, or 0
     */
    public int compareToById (Task other) {
        if (this.getIdentifier() > other.getIdentifier()) {
            return 1;
        } else if (this.getIdentifier() < other.getIdentifier()) {
            return -1;
        } else {
            return 0;
        }
    }

    /**
     * Returns the strength of the priority, so that
     * they can be compared numerically. The lower the
     * number, the higher the priority.
     * @return a number corresponding to the priority strength of this task
     */
    private int getPriorityStrength () {
        switch (this.getPriority()) {
            case 'U':
                return 1;
            case 'N':
                return 2;
            case 'L':
                return 3;
            default:
                return 0;
        }
    }

}
```

```java
/**
package edu.truman.cs260.krewson.toodle;

import java.util.ArrayList;

/**
 * Represents a list of multiple Task objects,
 * and contains methods to create, discard, and
 * generally act upon these tasks.
 * @author Tanner
 *
 */
public class TaskList {

    /**
     * Contains the list of tasks.
     */
    private ArrayList<Task> tasks = new ArrayList<Task>();

    /**
     * Provides an empty list of tasks.
     */
    public TaskList () {}

    /**
     * Provides a new TaskList object containing
     * the tasks passed.
     * @param tasks the tasks to add to the TaskList object
     */
    public TaskList (ArrayList<Task> tasks) {
        this.tasks = tasks;
    }

    /**
     * Cancels the given task, and sets a reason.
     * @param idOfTaskToCancel the id of the task to cancel
     * @param cancellationReason the reason the task was cancelled
     */
    public void cancelTask (int idOfTaskToCancel, String cancellationReason) {
        Task taskToCancel = this.getTaskById(idOfTaskToCancel);
        this.removeTask(taskToCancel);
        this.addTask(new CancelledTask(
                taskToCancel.getIdentifier(),
                taskToCancel.getDescription(),
                taskToCancel.getPriority(),
                taskToCancel.getOrder(),
                cancellationReason));
    }
```

```
/**
 * Completed the given task, and sets the completion date to
 * the date in which the methods is called.
 * @param idOfTaskToComplete the id of the task to complete
 */
public void completeTask (int idOfTaskToComplete) {
    Task taskToComplete = this.getTaskById(idOfTaskToComplete);
    this.removeTask(taskToComplete);
    this.addTask(new CompletedTask(
            taskToComplete.getIdentifier(),
            taskToComplete.getDescription(),
            taskToComplete.getPriority(),
            taskToComplete.getOrder(),
            new Date()));
}

/**
 * Gets all of the tasks in this TaskList object in
 * the order of their priority.
 * @return a list of tasks in order.
 */
public ArrayList<Task> getAllTasksInOrder () {
    this.sortTasksByPriorityOrder();
    return this.tasks;
}

/**
 * Gets a unique id that can be used to construct
 * new tasks.
 * @return the new id
 */
public int getNextId () {
    this.sortTasksById();
    if (this.tasks.size() > 0) {
        return this.tasks.get(this.tasks.size() - 1).getIdentifier() + 1;
    } else {
        return 0;
    }

}

/**
 * Creates a new task with the given descriptors,
 * and returns its id.
 * @param taskDescription the name of the task
 * @param priority urgent, normal, or low priority, denoted U, N, or L
 * @param taskOrder the priority of the task within the three priority
```

```
levels
     * @return the id of the newly created task
     */
    public int createNewTask(String taskDescription, char priority, int
taskOrder) {
        int newId = this.getNextId();
        this.addTask(new IncompleteTask(newId, taskDescription, priority,
taskOrder));
        return newId;
    }

    /**
     * Returns a list of tasks contained in this TaskList
     * object that are of the Incomplete type.
     * @return a list of incomplete tasks
     */
    public ArrayList<Task> getAllIncompleteTasks () {
        ArrayList<Task> newTaskList = new ArrayList<Task>();
        for (Task task : this.getAllTasksInOrder()) {
            if (task.STATUS.equals("Incomplete")) {
                newTaskList.add(task);
            }
        }
        return newTaskList;
    }

    /**
     * Returns a list of tasks contained in this TaskList
     * object that are of the Incomplete type and are contained
     * within a specified priority, U, N, or L.
     * @param priority the priority of the desired tasks
     * @return a list of incomplete tasks of the specified priority
     */
    public ArrayList<Task> getAllIncompleteTasksWithPriority (char priority) {
        ArrayList<Task> newTaskList = new ArrayList<Task>();
        for (Task task : this.getAllTasksInOrder()) {
            if (task.STATUS.equals("Incomplete") && task.getPriority() ==
priority) {
                newTaskList.add(task);
            }
        }
        return newTaskList;
    }

    /**
     * Adds the given task to this TaskList.
     * @param newTask the task to add
     */
```

```
private void addTask(Task newTask) {
    this.tasks.add(newTask);
}

/**
 * Removed the given task from this TaskList.
 * @param taskToRemove the task to remove.
 */
private void removeTask (Task taskToRemove) {
    this.tasks.remove(taskToRemove);
}

/**
 * Receives an id, and returns the corresponding
 * task with that id.
 * @param id the id of the task
 * @return the task with the given id
 */
private Task getTaskById (int id) {
    for (Task task : tasks) {
        if (task.getIdentifier() == id) {
            return task;
        }
    }
    return null;
}

/**
 * Sorts this TaskList's list of tasks by their ids.
 */
private void sortTasksById () {
    Collections.sort(this.tasks, new Comparator<Task>() {
        public int compare(Task task1, Task task2)
        {
            return task1.compareToById(task2);
        }
    });
}

/**
 * Sorts this TaskList's list of tasks by their priorities.
 */
private void sortTasksByPriorityOrder () {
    Collections.sort(this.tasks, new Comparator<Task>() {
        public int compare(Task task1, Task task2)
        {
            return task1.compareToByOrder(task2);
        }
        });
    }
}
```

```java
/**
package edu.truman.cs260.krewson.toodle;

import java.io.File;

/**
 * Wraps the file that is used to read in previously stored
 * data, and to write the information stored in memory once
 * the program is closed.
 * @author Tanner
 *
 */
public class ToodleFile {

    /**
     * Objects for reading in the file.
     */
    private FileReader fileStream;
    private Scanner fileIn;

    /**
     * Writes to the text file.
     */
    private PrintWriter fileOut;

    /**
     * Writes the given tasks to the text file.
     * @param tasksToWrite the tasks to write to the file.
     * @throws FileNotFoundException the text file is not found
     */
    public void writeTasksToFile (ArrayList<Task> tasksToWrite) throws
FileNotFoundException {
        fileOut = new PrintWriter("Task_List.txt");

        fileOut.println(tasksToWrite.size());
        fileOut.println();

        for (Task task : tasksToWrite) {
            fileOut.println(task.STATUS);
            fileOut.println(task.getIdentifier());
            fileOut.println(task.getDescription());
            fileOut.println(task.getPriority());
            fileOut.println(task.getOrder());

            if (task instanceof CompletedTask) {
                CompletedTask newTask = (CompletedTask)task;
                Calendar cal = Calendar.getInstance();
                cal.setTime(newTask.getCompletionDate());
                fileOut.println(cal.get(Calendar.YEAR));
                fileOut.println(cal.get(Calendar.MONTH));
                fileOut.println(cal.get(Calendar.DAY_OF_MONTH));
            } else if (task instanceof CancelledTask) {
                CancelledTask newTask = (CancelledTask)task;
                fileOut.println(newTask.getCancellationReason());
            }

            //add single line gap between tasks
            fileOut.println();
        }

        fileOut.close();
    }

    /**
     * Reads the previously written tasks from the text file,
     * places them in the correct objects, and returns a list of
     * the tasks that were read in.
     * @return the list of tasks read in
     * @throws IOException the text file could not be read from
     */
    public ArrayList<Task> readTasksFromFile () throws IOException {
        File file = new File("Task_List.txt");

        if (!file.isFile() && !file.createNewFile())
        {
            throw new IOException("Could not read from file Task_List.txt");
        }
        fileStream = new FileReader(file);
        fileIn = new Scanner(fileStream);

        ArrayList<Task> newTaskList = new ArrayList<Task>();

        int numberOfTasks;
        try {
            numberOfTasks = fileIn.nextInt();

            //move cursor to next line, and skip one
            fileIn.nextLine();
            fileIn.nextLine();
        } catch (Exception e) {
            numberOfTasks = 0;
        }

        for (int i = 0; i < numberOfTasks; i++) {
            String status = fileIn.nextLine();
            int identifier = fileIn.nextInt();
            fileIn.nextLine();
            String description = fileIn.nextLine();
            char priority = fileIn.nextLine().charAt(0);
            int order = fileIn.nextInt();
            fileIn.nextLine();

            if (status.equals("Completed")) {
                int year = fileIn.nextInt();
                int month = fileIn.nextInt();
                int day = fileIn.nextInt();
                fileIn.nextLine();
                Date completionDate = new GregorianCalendar(year, month,
day).getTime();
                newTaskList.add(new CompletedTask(identifier, description,
priority, order, completionDate));
            } else if (status.equals("Cancelled")) {
                String cancellationReason = fileIn.nextLine();
                newTaskList.add(new CancelledTask(identifier, description,
priority, order, cancellationReason));
            } else {
                newTaskList.add(new IncompleteTask(identifier, description,
priority, order));
            }

            //if its not the last task
            if (i != numberOfTasks - 1) {
                //account for single line gap between each task
                fileIn.nextLine();
            }

        }

        fileIn = null;
        fileStream.close();

        return newTaskList;
    }
}
```