Assignment 9

Refer to Canvas for assignment due dates for your section.

Objectives:

- Refactor earlier work to improve modularity.
- Design a solution with MV* architecture.
- Develop sortable custom objects.
- Use design patterns to solve common problems.

General Requirements

Create a new Gradle project for this assignment in your ***group*** GitHub repo.

For this assignment, please continue to use packages as in the past. There is only one problem so you may have only one package, but it can be helpful to create additional packages to keep your code organized. The requirements for repository contents are the same as in previous assignments.

New: Please include a ***brief*** write-up of which design pattern(s) you used, where, and why. This is just to help your grader understand your approach.

GitHub and Branches

Each individual group member should create their own branch while working on this assignment. Only merge to the master branch when you're confident that your code is working well. You may submit pull requests and merge to master as often as you like. Guidelines on working with branches are available from the assignment page in Canvas.

To submit your work, push it to GitHub and create a release on your master branch. Only one person needs to create the release.

The problem: TODO application

For this assignment, you may reuse any relevant code for command line parsing and file reading/writing from Assignment 8. If your design for Assignment 8 was sufficiently modular, you should find you can re-use some of your classes for this portion of the assignment with little modification. In codewalk, be prepared to talk about how well your original code for command line parsing and file processing supported extension and reuse. If you have to make substantial changes, be prepared to discuss how you improved your design for this assignment.

Your task is to build a command-line TODO application. The system will allow a user to add and track the status of their TODOs by due date, category, priority, and status (complete/incomplete).

The application stores all TODOs in a CSV file. The CSV file is a plain text file, containing data organized into columns separated by a comma. The data in each column is enclosed in double quotes. The first line of the file contains the headers for each column.

The CSV has 6 columns named `id`, `text`, `completed`, `due`, `priority`, and `category`. You can assume that the CSV column names will not change.

```
"id","text","completed","due","priority","category"
"1","Finish HW9","false","4/26/2022","1","school"
"2","Mail passport application, photo","true","5/31/2022","?","?"
```

The information in each column is enclosed in double quotes. It is possible that column entries may contain a comma. For example, on row 3 in the listing above, “Mail passport application, photo” is one valid piece of information, not two.

Some columns are considered optional and may not contain data. A cell that contains only “?” means that there is no value for this cell (you should not store “?” as a value in your objects). You can see an example on the last line in the listing.

A sample file containing a small number of todos, is available on Canvas, along with these instructions.

Todo data structure

A `Todo` consists of the following information:

- `text` - a description of the task to be done. This field is required.
- `completed` - indicates whether the task is completed or incomplete. If not specified, this field should be false by default. However, it should be possible to create a new `Todo` with `completed` set to true.
- `due` - a due date. This field is optional.
- `priority` - an integer indicating the priority of the todo. This field is optional. If the user chooses to specify a priority, it must be between 1 and 3, with 1 being the highest priority. If no priority is specified, the todo can be treated as lowest priority (i.e., 3).
- `category` - a user-specified String that can be used to group related todos, e.g., “school”, “work”, “home”. This field is optional.

Once a `Todo` is created, all fields should be immutable, with the exception of `completed`. In the system, each `Todo` will also have an integer ID that the user can use to update the

completion status of an individual todo (see below). You may choose how and where to generate and track the ID. Generating the ID should not be the user's responsibility.

Functionality

The system must support the following functionality:

- **Add a new todo.** The user must supply the information required by the `Todo` data structure. They can also choose to specify the optional information. When a new `Todo` is added, the CSV file should be updated.
- **Complete an existing todo.** The user set the completed status of an existing `Todo` to true. When the status is changed, the CSV file should be updated.
- **Display todos.** The user can request that the program display a list of `Todos`. You may choose how to format the list, but make sure it's user friendly! By default, all of the `Todos` should be printed but they can supply additional arguments to customize the list:
    - Filter the list to only include incomplete `Todos`;
    - Filter the list to only include `Todos` with a particular category;
    - Sort the `Todos` by date (ascending) ;
    - Sort the `Todos` by priority (ascending).

  The two filter arguments can be combined but only one sort can be applied at a time. For example, the user could request all incomplete `Todos` with category, "work", sorted by date but the user cannot request all incomplete `Todos` sorted by date AND priority.

When sorting by date, there may be Todos that do not have a due date. These Todos should come after all Todos that do have a due date.

Your program should accept the following command line arguments in <u>any</u> order:

`--csv-file <path/to/file>`	`The CSV file containing the todos. This option is required.`
`--add-todo`	`Add a new todo. If this option is provided, then --todo-text must also be provided.`
`--todo-text <description of todo>`	`A description of the todo.`
`--completed`	`(Optional) Sets the completed status of a new todo to true.`
`--due <due date>`	`(Optional) Sets the due date of a new todo. You may choose how the date should be formatted.`
`--priority <1, 2, or 3>`	`(Optional) Sets the priority of a new todo. The value can be 1, 2, or 3.`

`--category <a category name>`	(Optional) Sets the category of a new todo. The value can be any String. Categories do not need to be pre-defined.
`--complete-todo <id>`	Mark the Todo with the provided ID as complete.
`--display`	Display todos. If none of the following optional arguments are provided, displays all todos.
`--show-incomplete`	(Optional) If --display is provided, only incomplete todos should be displayed.
`--show-category <category>`	(Optional) If --display is provided, only todos with the given category should be displayed.
`--sort-by-date`	(Optional) If --display is provided, sort the list of todos by date order (ascending). Cannot be combined with --sort-by-priority.
`--sort-by-priority`	(Optional) If --display is provided, sort the list of todos by priority (ascending). Cannot be combined with --sort-by-date.

A user can request the program perform all three tasks (add, complete, and display) in one run of the program. They may only add one `Todo` at a time, but they may complete multiple `Todos` by repeating the `--complete-todo` option for each `Todo` to complete. E.g. `--complete-todo 5 --complete-todo 2` would complete the `Todo` with ID 5 and the `Todo` with ID 2.

When a user provides an illegal combination of inputs, the program should exit with a helpful error message, and a short explanation of how to use the program along with examples. If the user attempts to complete a `Todo` that does not exist or display a subset of `Todos` that returns no results, the program should also provide feedback to the user.