

DEADLINE

1

SE lecture  
@icube #lecture

03-14 15:00

EVENT

2

SE lecture  
@icube #lecture

02-04 16:00

03-13 24:00

FLOATING

3

SE lecture  
@icube #lecture

RESERVED

4

SE lecture

Apr 1 13:00

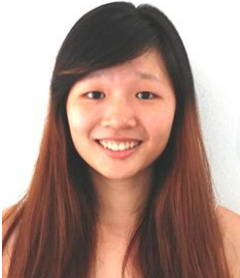
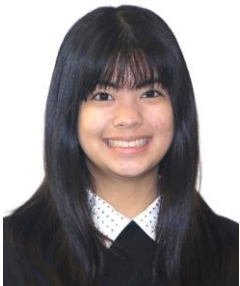


Apr 3 13:00

Apr 4 13:00

type your command here

## J.Listee

Supervisor: Karan Kamath Extra feature: GoodGui

 <p>Her Kai Lin</p> <p>Team lead, Scheduling and tracking</p>	 <p>Chloe Odquier Fortuna</p> <p>Git expert, Integration</p>	 <p>Boh Tuang Hwee, Jehiel</p> <p>Code quality</p>	 <p>Zhu Bingjing</p> <p>Documentation and UI</p>
--	---	--	---

**Credit**

## **ACKNOWLEDGEMENTS OF THIRD PARTY MATERIALS**

1. J. Listee transfers data between Java and JavaScript using JSON (JavaScript Object Notation), which is a lightweight data-interchange format.

## User guide

# TABLE OF CONTENTS

## Quick start

## Feature details

- ❖ Adding tasks
- ❖ Updating tasks
- ❖ Postponing tasks
- ❖ Deleting tasks
- ❖ Searching tasks
- ❖ Reserving time slots
- ❖ Confirming time slots
- ❖ Marking as done/undone
- ❖ Undoing/redoing
- ❖ Exiting

## Cheat sheet

## QUICK START

---

1. Configure java environment on your computer. You can download JDK [here](#).
2. Run J.Listee.jar
3. Try to type the following commands in the command box and see what's happening:

add my first task @icube (Feb 14) #cs2103

update 1 updated first task

done 1

undo

postpone 1 2

show #cs2103

delete 1

Now you can add more tasks and start to manage them using J.Listee!

You can also use the cursor keys (↑ and ↓) to scroll through the tasks and select a specific task.

### **Note:**

In this document, we use the following syntax to describe the command formats:

<some command element> **e.g.** <task description>

[<the command element inside square brackets is optional>] **e.g.** [<optional start time>]

## FEATURE DETAILS

### ✧ Adding tasks

#### **Format:**

add <task description> [[(<start time> -] <end time>)] [@ location] [#<tag>[#<tag> [...]]]

#### **Description:**

This command enables you to add a task into J.Listee. The task will be assigned a task number when listed to you. After adding the task, the list of all tasks will automatically be displayed in order of urgency (overdue, today, tomorrow, timed, untimed, reserved).

#### **Must-have arguments:**

Task description

#### **Optional arguments:**

start time, end time, location, tags (tags cannot contain whitespace)

### ✧ Updating tasks

#### **Format:**

update <task number> [<new task description>] [- ()] [[[<new start time> -] <new end time>]] [-@] [@ new location] [-#<existing tag> [#<new tag> [...]]]

#### **Description:**

This command enables you to update an existing task from J.Listee.

#### **Must-have arguments:**

a task number indicating the specific task

### **Optional arguments:**

new task description, removing any time, new start time, new end time, removing existing location, new location, removing specific tags, new tags.

## ✧ Postponing tasks

### **Format:**

Postpone <task number> <number of days>

### **Description:**

This command enables you to postpone the deadline of an existing task for some days.

### **Must-have arguments:**

task number and the number of days

### **Shortcut:**

Select the specific task and press *p*, the deadline of the task will be postponed for 1 day.

## ✧ Deleting tasks

### **Format:**

delete <1st task number> [, <2nd task number> [...]]

delete <task group> \*all/done/undone/overdue/today/tomorrow/reserved

### **Description:**

This command enables you to delete existing tasks from J.Listee.

**Must-have arguments:**

group name or at least one task number

**Optional arguments:**

more task number separated by English comma

**Shortcut:**

Select the specific task and press *Backspace* or *Delete*, the task will be deleted.

✧ **Searching tasks**

**Format:**

show <keywords> \*part of description, time, location, tags

show <task group> \*all/done/undone/overdue/today/tomorrow/reserved

show ([[<start time>-] <end time>])

**Description:**

This command enables you to see specific tasks in a time period.

**Must-have arguments:**

Keywords separated by whitespace/group name/an end time

**Optional arguments:**

Start time

## ✧ Reserving time slots

### **Format:**

reserve <task description> (<start time1>- <end time1>) [(<start time 2>- <end time2>)  
[<start time 3>- <end time3> [...]]] [@ location] [#<tag>[#<tag> [...]]]

### **Description:**

This command enables you to reserve multiple time slots for later use.

### **Must-have arguments:**

Task description, start time1, end time1

### **Optional arguments:**

Start time2, end time2, location, tags

## ✧ Confirming time slots

### **Format:**

confirm <task number> [new description] (<time slot number>) [-@] [@ new location] [-  
#<existing tag> [#<new tag> [...]]]

### **Description:**

This command enables you to confirm one of the time slots you have reserved and update its information if you need.

### **Must-have arguments:**

Task number, time slot number



### **Optional arguments:**

new task description, removing existing location, new location, removing specific tags, new tags.

## ✧ **Marking as done/undone**

### **Format:**

done <1st task number> [, <2nd task number> [...]]

done <task group> \*all/done/undone/overdue/today/tomorrow/reserved

undone <1st task number> [, <2nd task number> [...]]

undone <task group> \*all/done/undone/overdue/today/tomorrow/reserved

### **Description:**

This command enables you to mark existing tasks as done/undone

### **Must-have arguments:**

group name/at least one task number

### **Optional arguments:**

more task number separated by English comma

### **Shortcut:**

Select the specific task and press *d*, if the task is already marked as done, it will be marked as undone; otherwise it will be marked as done.

## ✧ Undoing/redoin

### **Format:**

undo

redo

### **Description:**

This command enables you to undo or redo the most recent operation.

### **Shortcut:**

Press Ctrl + z to undo the last operation. Press Ctrl + y to redo the last operation.

## ✧ Exiting

### **Format:**

exit

### **Description:**

This command enables you to exit.

### **Shortcut:**

Press Esc to exit

# CHEAT SHEET

#	Command	Example
1	add <task description> [[[<start time> -] <end time>]] [@ location] [#<tag>[#<tag> [...]]]	add Computer class (every Friday 16:00-18:00) @icube #lecture
2	update <task number> [<new task description>] [- ( )] [[[<new start time> -] <new end time>]] [-@] [@ new location] [-#<existing tag> [#<new tag> [...]]]	update 1 SE lecture  update 2 -@  update 3 -#lecture #assignment
3	Postpone <task number> <number of days>	postpone 2 3
4	delete <1st task number> [, <2nd task number> [...]]  delete <task group>	delete 1,2,3  delete today
5	show <keywords>  show <task group>  show ([[<start time>-] <end time>])	show #lecture, computer  show today  show (Apr 1-Apr 3)
6	reserve <task description> (<start time1>- <end time1>) [[(<start time 2>- <end time2>) [<start time 3>- <end time3> [...]]] [@ location] [#<tag>[#<tag> [...]]]	reserve meeting with boss (Tue 15:00-Tue 16:00) (Thu 15:00- Thu 16:00) @lt13 #meeting
7	confirm <task number> [new description] (<time slot number>) [-@] [@ new location] [-#<existing tag> [#<new tag> [...]]]	confirm 2 meeting with manager (2) -@
8	done/undone <1st task number> [, <2nd task number>	done 1,2,5

	<p>[...]</p> <p>done/undone &lt;task group&gt;</p>	<p>done today</p> <p>undone done</p>
<b>9</b>	<p>undo</p> <p>redo</p>	<p>undo</p> <p>redo</p>
<b>10</b>	<p>exit</p>	<p>exit</p>

*\*Group name: all/done/undone/overdue/today/tomorrow/reserved*

*\*Keywords: part of description, time, location, tags*

## Developer guide

# TABLE OF CONTENTS

## Architecture

### GUI component

- ❖ App Class
- ❖ GUIController Class
- ❖ AppPage Class

### Logic component

- ❖ Logic Class

### Storage component

- ❖ Storage Class
- ❖ LogStorage Class

### Parser component

- ❖ Parser Class

### History component

- ❖ History Class

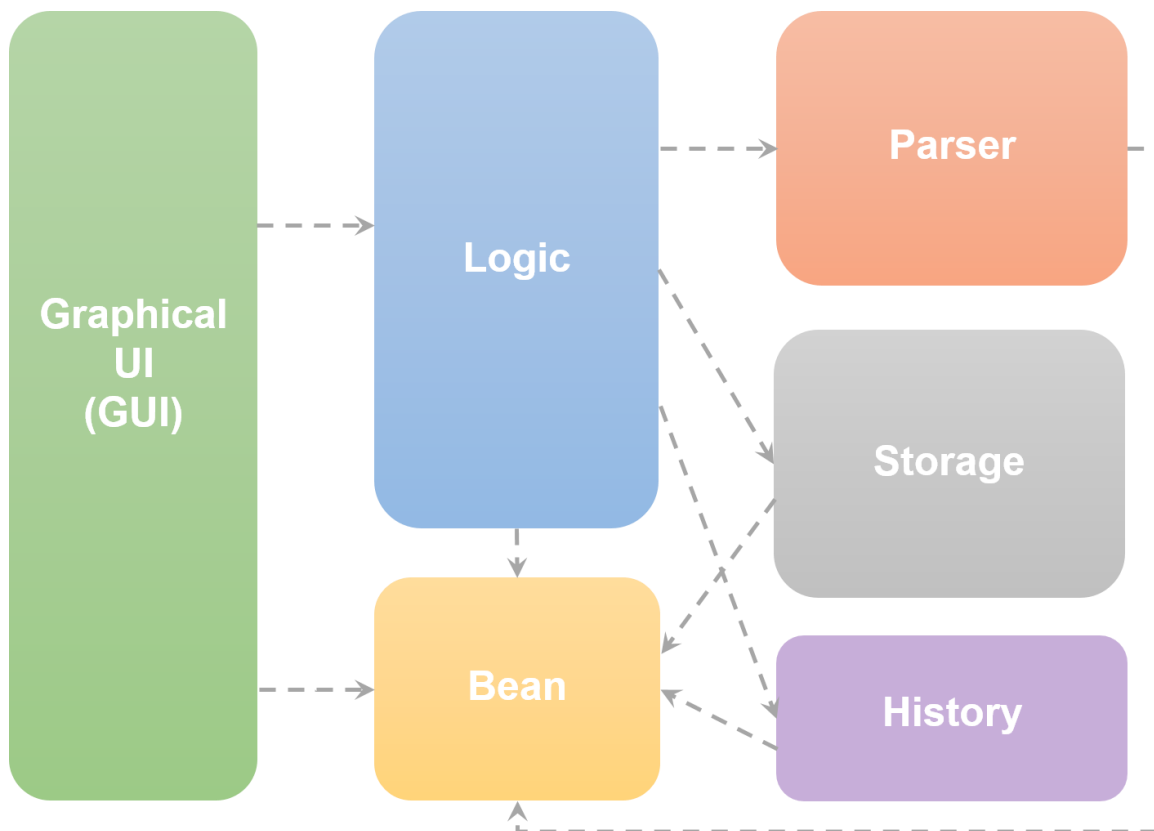
### Bean component

- ❖ Task Class

❖ Command Class

❖ Display Class

## ARCHITECTURE

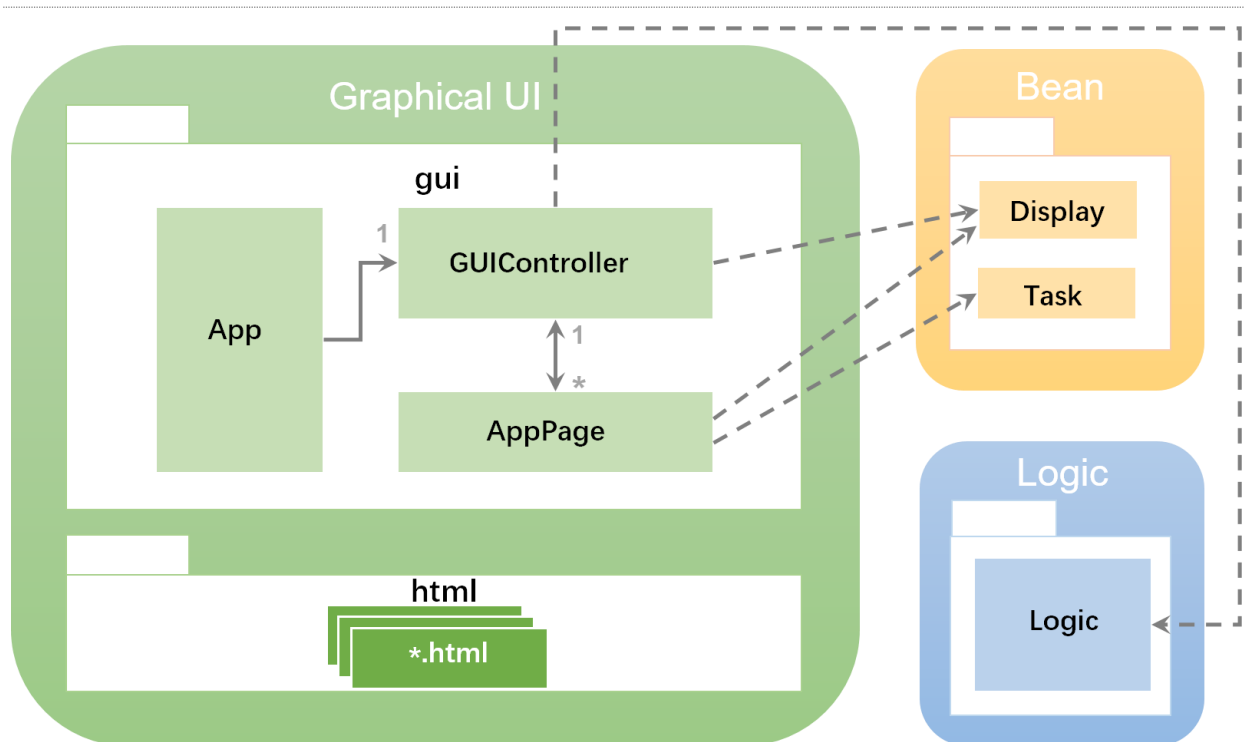


J.Listee is made up of six main components.

- ❖ The **GUI** component consists of JavaFX's HTML files which define the layout that users interact with and the Java files which communicate with these HTML files through JavaScript.
- ❖ The **Logic** component serves as the interface between GUI and the respective components.

- ❖ The **Storage** component consists of two parts. One is a task file which saves the user's tasks so that he can access his to-do list whenever he starts J.Listee. The other is a log file which saves the storage location of user's task file.
- ❖ The **Parser** component consists of a parser class which is used to analyze user's command and then return the command to Logic for following execution.
- ❖ The **History** component is where the previous display objects are stored in case user wants to undo/redo certain commands.
- ❖ The **Bean** component represents objects involved in the management of tasks such as the tasks to do and commands that the user wants the app to execute.

## GUI COMPONENT



The **GUI** component is made up of two packages, **gui** and **html**. The **gui** package contains the Java files that control what users see while the **html** package contains HTML files that describe the layout that users interact with.

## ❖ App Class

The **App** class is the entry of J.Listee. **App** extends from JavaFX's **Application** class and overrides its start method. This method is the starting point of the whole application and very importantly, initializes the main frame that are required for the GUI. This method also calls another method to judge whether it's necessary for user to choose storage location and show corresponding page according to the judgement.

## ❖ GUIController Class

The **GUIController** class is the main driver for the **GUI** component. It controls what users see and handles user inputs by passing them to the Logic component.

### Notable APIs:

Return Type	Method and Description
void	<b>createFile(String userChosenfile)</b> : Call Logic to write into the log file the user chosen storage location and create the text file that holds the contents of the task list for storage.
void	<b>initializeList(Stage stage, String filePath)</b> : Initialize the start page which display deadlines, events and floating tasks
void	<b>displayWelcome (Stage stage)</b> : Set the current scene of stage to welcome page.
void	<b>displayList (Stage stage, Display display)</b> : Set the current scene of stage to showList page.
void	<b>handelUserInput(String command)</b> : Decides what to do when command is entered.

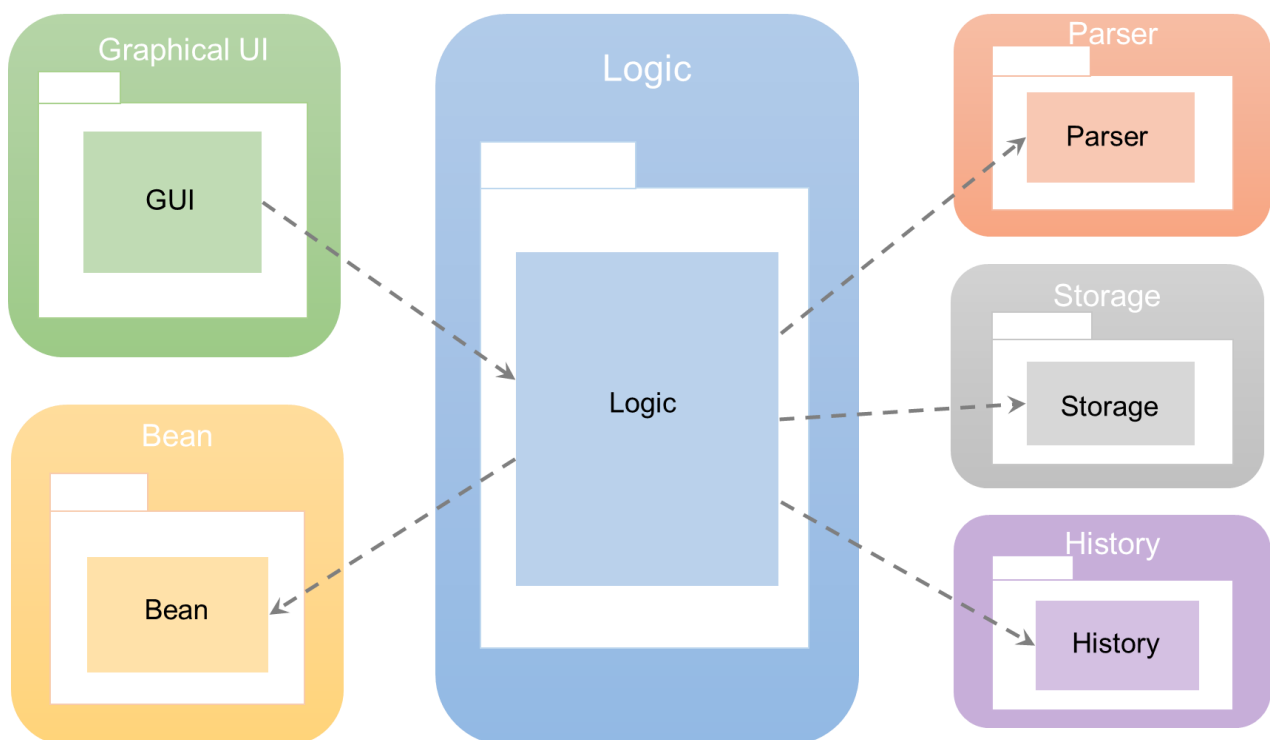


## ❖ **AppPage Class**

The **AppPage** class is an abstract class which extends from JavaFX's **Pane** class. Each child of **AppPage** will load a html file from **html** package, and has an inner class as its JavaScript interface object which acts as a bridge between Java and JavaScript.

This class has a reference to **GUIController** and calls some methods of **GUIController**. For example, it will call **handelUserInput** whenever user inputs command. This ensures that the logic is handled by **GUIController** to avoid unnecessary coupling between **AppPage** and **Logic**.

## LOGIC COMPONENT



The **Logic** component serves as the interface between **GUI** and the respective components.

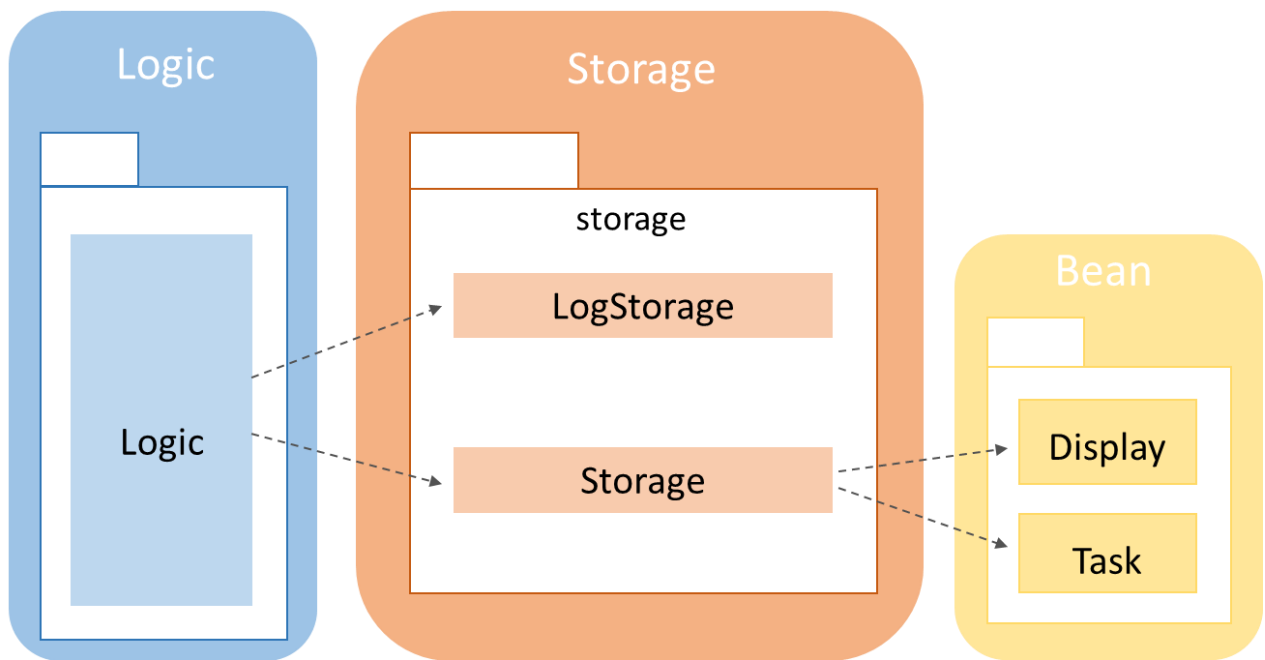
## ❖ Logic Class

**Logic** serves as the interface between GUI and the respective components.

### Notable APIs:

Return Type	Method and Description
void	<b>createFile(String filepath)</b> : This method is called on the very first startup of the program. The file path, which is specified by the user, is then sent to storage. The result is a Boolean which indicates whether the file is created successfully.
Display	<b>initializeProgram(String filePath)</b> : initializeProgram is called every start up the program to retrieve the display from storage.
Display	<b>ExecuteUserCommand(String userInput)</b> : The method is called by GUI whenever the user enters in a command. User inputs are sent to the parser to be processed. The result is a command which is then executed at Logic. All commands implement the command interface. Logic then sends the output to History and/or Storage to be saved if necessary before returning it to GUI.

## STORAGE COMPONENT



The **Storage** Component composes of two classes: **LogStorage** and **Storage**. This component saves the user's tasks so that he can access his to do list whenever he starts J.Listee.

### ❖ LogStorage Class

The **LogStorage** class is used for reading and writing the log file which contains the file path of the text file used as storage. When first starting up J.Listee, **LogStorage** writes the location of the storage into the log file. Upon subsequent uses of J.Listee, **LogStorage** automatically reads the log file for the storage's location.

#### Notable APIs:

Return Type	Method and Description
String	<b>readLog()</b> : Reads the log file that contains the filepath of the task list. If the log file doesn't exist, creates the log file.

void	<b>writeLogFile(String filePath)</b> : Writes the filepath into the log file.
------	---

## ❖ Storage Class

The **Storage** class is used for reading and writing to the human-readable and editable text file which contains the storage of all the tasks used in J.Listee. It retrieves the data stored in the text file for Logic to manipulate upon opening up J.Listee, and saves the changes to the file the user has created after operations that update the task list.

### Notable APIs:

Return Type	Method and Description
void	<b>createFile(String filepath)</b> : Creates the text file that holds the contents of the task list for storage.
Display	<b>getDisplay(String filepath)</b> : Reads the contents of the storage file and sends it as a Display object for the Logic component.
void	<b>saveFile(Display thisDisplay)</b> : Writes all the contents of the Display object to the text file containing the storage of the task list.
Storage	<b>getInstance()</b> : Instantiates the Storage class according to the Singleton pattern

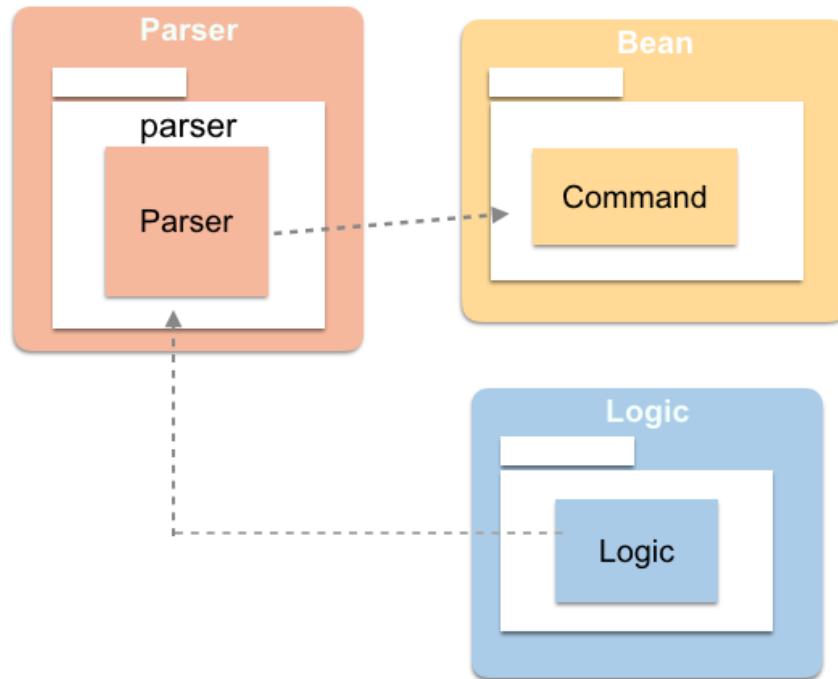
### Patterns:

Singleton – Only one instance of the Storage class is needed. Instead of using a public constructor to instantiate this class, the method `getInstance()` is used to prevent more than one instance of Storage being created.

### Principles:

Single Responsibility Principle – Every class has its own responsibility. Therefore, LogStorage only reads and writes to the log file, and Storage only reads and writes to the text file that contains the task list.

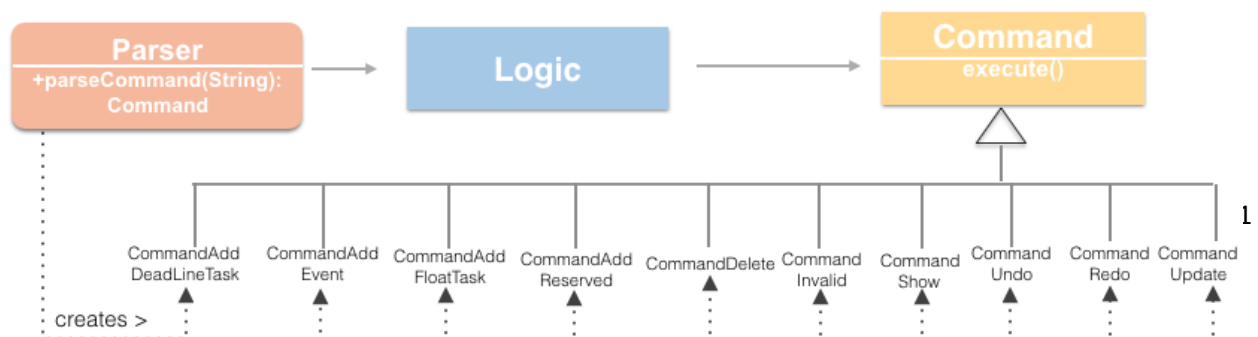
## PARSER COMPONENT



The **Parser** component consists of a **Parser** class which is used to analyze user's command and then return the command to Logic for following execution.

### ❖ Parser Class

The **Parser** class objective is to break down the input string of the user and determine type of command, creates a **Command** object and passes to the **Logic** class for execution.



**Notable APIs:**

Return Type	Method and Description
Command	<b>parseCommand(String inputLine)</b> : Returns the Command object with defined fields in each Command object depending the type of command.

**Example:**

User input: “add CS2103 V0.1 (20/3/16 23:59) @online #work”

Parser creates **Command** object, noting that it is a deadline task as it only consists of date and time, hence initialising taskDescription, Location, Date, Time, Tags.

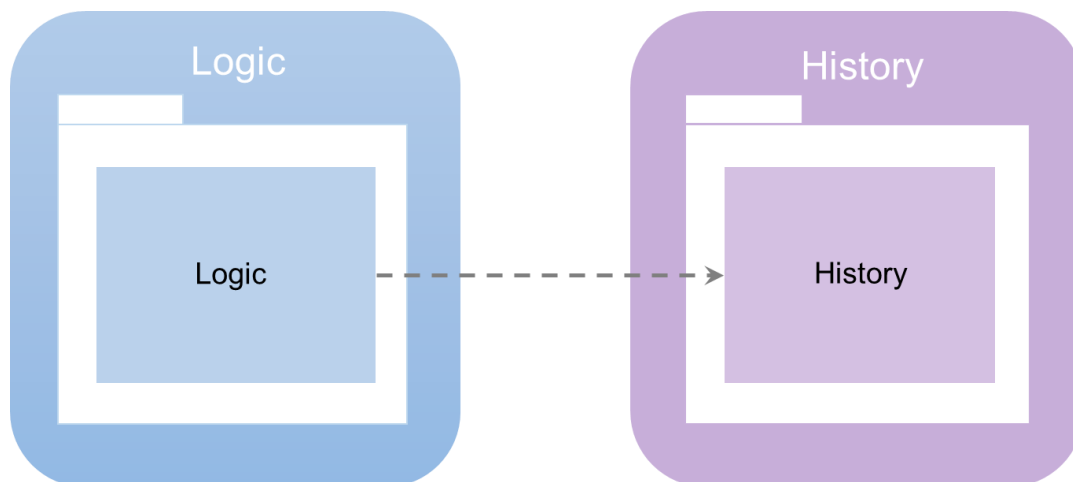
```
Command deadLineTask = new
CommandAddDeadlineTask(taskDescription, location, endDateTime,
tagLists);
```

```
return deadLineTask;
```

These fields can be access by **Logic** to execute the command properly.

## HISTORY COMPONENT

---



**History** is where the previous display objects are stored in case user wants to undo/redo certain commands.

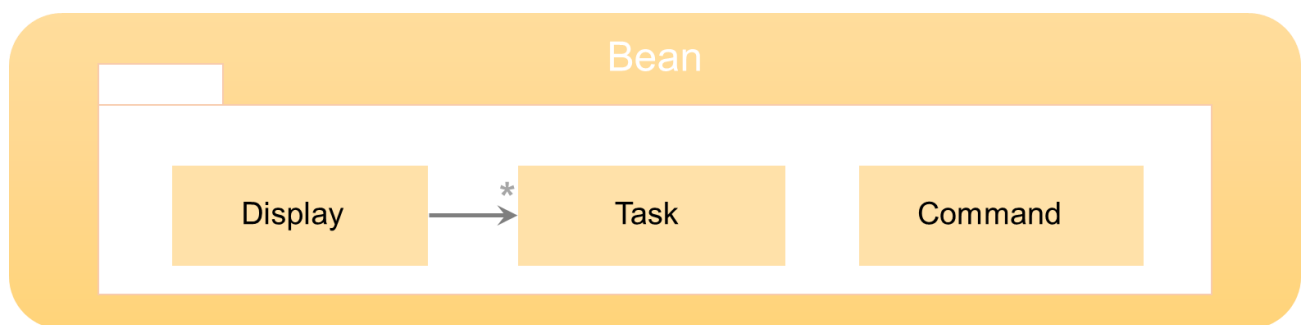
### ❖ History Class

The **History** Class is where the previous display objects are stored in case user wants to undo/redo certain commands.

**Notable APIs:**

Return Type	Method and Description
void	<b>saveDisplay(Display display)</b> : Saves the argument display into a list of Display objects.
Display	<b>getDisplay(int offset)</b> : This method is to facilitate the undo/redo commands. An offset of -1 will return the previous display while an offset of 1 will return the next display (only possible if user has previous issued an undo command)

## BEAN COMPONENT



The **Bean** component contains the classes that represent the various elements that are required in managing user's tasks.

**Logic** manipulates these classes and all the other components will use the data within these classes to do their jobs.

### ❖ **Display Class**

This class represents information that needs to be displayed on the app. Every **Display** object will have several lists of different tasks and a message.

### ❖ **Task Class**

This class represents user's tasks. It's a generic object and has four types of children: **TaskDeadline, TaskEvent, TaskFloat, TaskRsaved**.

### ❖ **Command Class**

This class represents user's commands. When user enter some commands, Parser will analyze it and create a Command object. It's a generic object and has ten types of children: **CommandAddDeadlineTask, CommandAddEvent, CommandAddFloatTask, CommandAddReserved, CommandDelete, CommandInvalid, CommandRedo, CommandShow, CommandUndo, CommandUpdate**.