# Setup

## \_workingList and \_workingListMaster

QLLogic uses working lists to keep track to the tasks that are in QuickLyst. \_workingList holds the tasks that needs to be passed to QLGUI to be displayed to the user. \_workingListMaster holds all the tasks that has been added but not deleted by the user. Hence is a \_workingList is a subset of \_workingListMaster, and they both hold the references to the same Task when the task is present in both lists.

## \_undoStack and \_redoStack

QLLogic uses stacks to perform undo and redo operations. \_undoStack is used to store previous versions of the working lists and \_redoStack is used to store versions of the working lists that are ahead of the current working lists. Section \_\_\_ will cover on how the undo and redo operations work.

The following object diagram illustrate the relationship between Tasks, working lists, and stacks.



## Code fragments

During setup, working lists have to be loaded with previously saved tasks and the \_undoStack needs to be hold the first version of the working lists. The setup method code is shown below.

**public** **static** LinkedList<Task> setup(String fileName) {

*\_fileName* = fileName;

*\_undoStack* = **new** Stack<LinkedList<Task>>();

*\_redoStack* = **new** Stack<LinkedList<Task>>();

*\_workingList* = QLStorage.*loadFile*(fileName);

*\_workingListMaster* = **new** LinkedList<Task>();

*copyList*(*\_workingList*, *\_workingListMaster*);

*\_undoStack*.push(*\_workingListMaster*);

*\_undoStack*.push(*\_workingList*);

**return** *\_workingList*;

}

# Important Algorithms

## Sorting tasks

Bubble sort is used when sorting \_workingList as it is simple and stable sorting algorithm. A stable algorithm is needed as the relative position of tasks from the previous sort must be preserved when the tasks are sorted again by the next criteria. This is to ensure that the eventual result of the multiple criteria sort according to the specified criteria is sorted at all levels. To sort by multiple criteria, the tasks is sorted by the lowest level criteria first, followed by higher level criteria in the next iterations. The following example illustrates the idea.

*Command: sort –d A –s A –p D*

|  |  |  |  |
| --- | --- | --- | --- |
| Task Name | Due | Start | Priority |
| Task 5 | 14/3 | 28/2 | H  By descending priority |
| Task 2 | 13/3 | 1/3 | M |
| Task 6 | 14/3 | 1/3 | L |
| Task 4 | 13/3 | 2/3 | L |
| Task 7 | 14/3 | 1/3 | L |
| Task 3 | 13/3 | 2/3 | M |
| Task 1 | 13/3 | 1/3 | H |

|  |  |  |  |
| --- | --- | --- | --- |
| Task Name | Due | Start | Priority |
| Task 5 | 14/3 | 28/2 | H |
| Task 1 | 13/3 | 1/3 | H |
| Task 2 | 13/3 | 1/3 | M |
| Task 3 | 13/3 | 2/3 | M |
| Task 6 | 14/3 | 1/3 | L |
| Task 4 | 13/3 | 2/3 | L |
| Task 7 | 14/3 | 1/3 | L |

By ascending start date

|  |  |  |  |
| --- | --- | --- | --- |
| Task Name | Due | Start | Priority |
| Task 5 | 14/3 | 28/2 | H |
| Task 1 | 13/3 | 1/3 | H |
| Task 2 | 13/3 | 1/3 | M |
| Task 6 | 14/3 | 1/3 | L |
| Task 7 | 14/3 | 1/3 | L |
| Task 3 | 13/3 | 2/3 | M |
| Task 4 | 13/3 | 2/3 | L |

|  |  |  |  |
| --- | --- | --- | --- |
| Task Name | Due | Start | Priority |
| Task 1 | 13/3 | 1/3 | H |
| Task 2 | 13/3 | 1/3 | M |
| Task 3 | 13/3 | 2/3 | M |
| Task 4 | 13/3 | 2/3 | L |
| Task 5 | 14/3 | 28/2 | H |
| Task 6 | 14/3 | 1/3 | L |
| Task 7 | 14/3 | 1/3 | L |

By ascending due date

### Code example

**private** **static** LinkedList<Task> executeSort(String fieldLine,

StringBuilder feedback) {

LinkedList<String> fields = CommandParser.

*processFieldLine*(fieldLine);

**if**(fields.size() == 0) {

feedback.append("No field entered.");

**return** *\_workingList*;

}

LinkedList<**char**[]> sortingCriteria = CommandParser.

*getSortingCriteria*(fields);

*sortByCriteria*(sortingCriteria, feedback);

*updateUndoStack*();

**return** *\_workingList*;

}

## Finding tasks

To find tasks meeting a certain criteria, \_workingList is filtered each criterion in the order they are keyed in. The result is a \_workingList that contains only the tasks that meet the criteria. If the workingList is empty (i.e. no tasks found), the \_workingList is restored to its previous state. Using the tasks in the previous section, the following example illustrates the idea.

|  |  |  |  |
| --- | --- | --- | --- |
| Task Name | Due | Start | Priority |
| Task 1 | 13/3 | 1/3 | H |
| Task 2 | 13/3 | 1/3 | M |
| Task 3 | 13/3 | 2/3 | M |
| Task 4 | 13/3 | 2/3 | L |
| Task 5 | 14/3 | 28/2 | H |
| Task 6 | 14/3 | 1/3 | L |
| Task 7 | 14/3 | 1/3 | L |

|  |  |  |  |
| --- | --- | --- | --- |
| Task Name | Due | Start | Priority |
| Task 1 | 13/3 | 1/3 | H |
| Task 2 | 13/3 | 1/3 | M |
| Task 6 | 14/3 | 1/3 | L |
| Task 7 | 14/3 | 1/3 | L |

*Command: find –s 0103 –d 1403 –p M*

Match start date

Match due date

|  |  |  |  |
| --- | --- | --- | --- |
| Task Name | Due | Start | Priority |
| Task 6 | 14/3 | 1/3 | L |
| Task 7 | 14/3 | 1/3 | L |

Match priority

|  |  |  |  |
| --- | --- | --- | --- |
| Task Name | Due | Start | Priority |
|  |  |  |  |

Restore

### Code fragments

**private** **static** LinkedList<Task> executeFind(String fieldLine,

StringBuilder feedback) {

LinkedList<Task> workingListBackUp = **new** LinkedList<Task>();

*copyList*(*\_workingList*, workingListBackUp);

*recover*();

LinkedList<String> fields = CommandParser.

*processFieldLine*(fieldLine);

**for**(**int** i = 0; i < fields.size(); i++) {

*filterWorkingListByCriteria*(fields.get(i), feedback);

}

**if**(*\_workingList*.isEmpty() || fields.isEmpty()) {

feedback.append(***MESSAGE\_NO\_MATCHES\_FOUND***);

*\_workingList* = workingListBackUp;

**return** *\_workingList*;

}

*updateUndoStack*();

**return** *\_workingList*;

}

## Undo and Redo

Undo and redo function is implemented using an \_undoStack and \_redoStack. After each add/ edit/ delete/ complete operation, a copy of \_workingList and \_workingListMaster will be pushed onto \_undoStack as a “snapshot” of the state of the lists. Since the working lists contain Tasks which are objects, new Tasks are created with identical attributes as those in the working lists when copying the working lists so that they do not get affected by edit functions when they are in the stack. This is achieved using the copyListsForUndoStack() method. The diagram below illustrates this process.



When user calls undo, the “current” working lists are be popped out of \_undoStack and pushed into \_redoStack, and the working lists are referenced to the “previous” working lists on top of the \_undoStack. When the user calls redo, the “current” working lists are be popped out of \_redoStack and pushed into \_undoStack, and the working lists are referenced back to them. The diagrams below illustrate these processes.

*Undo operation*

*Redo operation*



### Code fragments

Insert code example