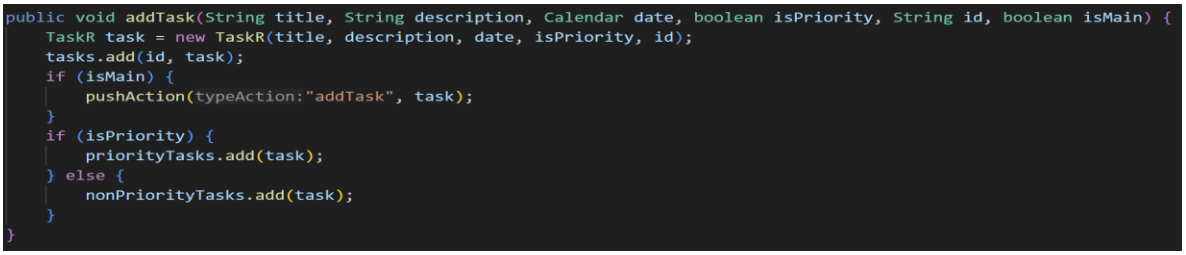
TIME & SPACE COMPLEXITY ANALYSIS

Method #1: addTask



```
public void addTask(String title, String description, Calendar date, boolean isPriority, String id, boolean isMain) {
    TaskR task = new TaskR(title, description, date, isPriority, id);
    tasks.add(id, task);
    if (isMain) {
        pushAction(typeAction:"addTask", task);
    }
    if (isPriority) {
        priorityTasks.add(task);
    } else {
        nonPriorityTasks.add(task);
    }
}
```

Time Complexity:

In order to analyse the time complexity of the addTask Method it is important to separate it into smaller operations:

- Creating the task:
  Creating a TaskR object with the input info given by the user is a constant time operation of (O(1)).
- Adding the task to tasks:
  Adding an objects to the tasks is O(1).
- Calling pushAction (if isMain = true):
  It's a time of O(1) constant time.
- Adding task to priorityTasks or nonPriorityTasks:
- Adding an objects to the tasks is also O(1).

Based on the individual analysis of each part of the operations used in the addTask Method, the time complexity for this method is O(1).

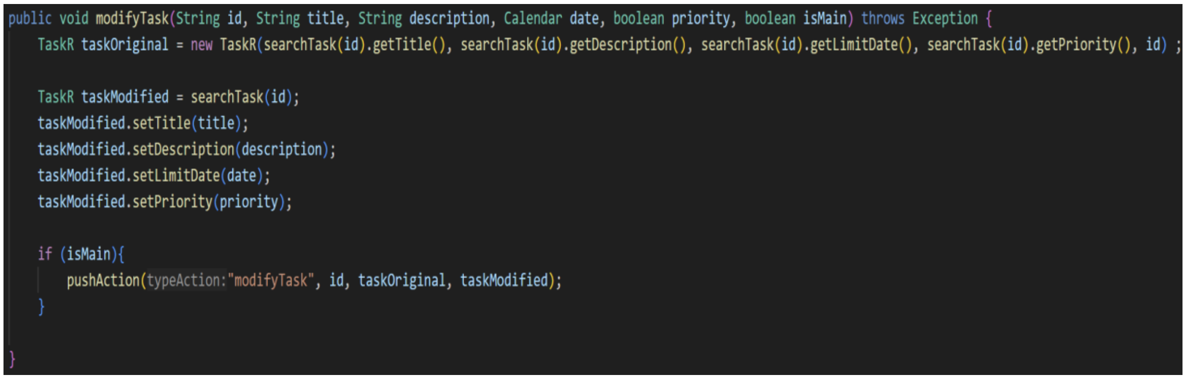
Space Complexity:

The space complexity of this method simpler, it will also be divided into the main operations:

- task:
  Creating a TaskR object consumes a constant amount of memory (O(1)).
- Adding the task to tasks:
  Also consumes a constant amount of memory(O(1)).
- Calling pushAction (if isMain = true):
  Consumes a memory of (O(1)).
- Adding task to priorityTasks or nonPriorityTasks:
  This addition does not affect much of the memory used, but it is also (O(1)).

The space complexity of the addTask method is O(1) because it just uses creating a new TaskR object and a few references to objects in memory, which are constant.

Method #2: modifyTask



```java
public void modifyTask(String id, String title, String description, Calendar date, boolean priority, boolean isMain) throws Exception {
    TaskR taskOriginal = new TaskR(searchTask(id).getTitle(), searchTask(id).getDescription(), searchTask(id).getLimitDate(), searchTask(id).getPriority(), id) ;

    TaskR taskModified = searchTask(id);
    taskModified.setTitle(title);
    taskModified.setDescription(description);
    taskModified.setLimitDate(date);
    taskModified.setPriority(priority);

    if (isMain){
        pushAction(typeAction:"modifyTask", id, taskOriginal, taskModified);
    }

}
```

Time Complexity:

This method is made of many different operations:

- Search for the task using searchTask(id) two times:
  The time to find the task with the given ID is determined by the data structure used for storing tasks. If it's a simple list, the time complexity is O(N), where N is the number of tasks.
- Modifying taskModified:
  For modifying attributes of taskModified the time is constant: (O(1)) because it's just assigning new values to its attributes.
- Calling pushAction:
  The first call uses pushing two TaskR objects onto the stack, which is O(1).
  The second call uses pushing two TaskR objects onto the stack, which is also O(1).
  This means that the pushAction calls used in modifyTask is O(1).

This gives as a result that the time complexity of the modifyTask method is a result of the search operation:

Time Complexity: O(N) (searching) + O(1) (pushAction) = O(N + 1) = O(N)

Space Complexity:

The space complexity of this method simpler, it will also be divided into the main operations:

- taskOriginal:
  Creating a new TaskR object to store the original state of the task consumes a constant amount of memory (O(1)) since it involves creating a new object with a fixed number of attributes.
- taskModified:
  Creating a reference to the TaskR object found through searchTask doesn't significantly increase memory usage. It's O(1) in terms of space complexity.

Then, the space complexity of modifyTask is O(1) because it uses a stable number of object references and doesn’t depend on the size of the input or any dynamic data structures.