# Assignment Report

## Requirements & limitations

Assignment 1 entails the challenge of implementing two functioning todo list programs each following a different programming paradigm, Imperative and Object-Oriented style respectively.

The todo list programs must store data in a queue (first-in-first-out). Each item in the queue can be either a ‘task’ or an ‘event’ wherein tasks consist of: a start time, duration and a list of people assigned to the task, while an event has a date, a start time and a location.

We must distinguish the two programming paradigms to better understand how to implement them. Imperative programming is a programming paradigm where the state of the program is changed using statements. On the other hand, object oriented programming paradigm is based upon the generation and usage of objects which contain data and methods which aims to incorporate the advantages of modularity and reusability.

For the imperative implementation it is forbidden to utilise any object oriented aspects both built-in within the chosen programming language and the creation of my own classes/objects. Common data structures such as arrays and hashmaps are also forbidden among many others thus ruling out the legality of data structures for this assignment. Since a todo list inherits the behaviour of a queue, we must find a way to implement one while following the restrictions.

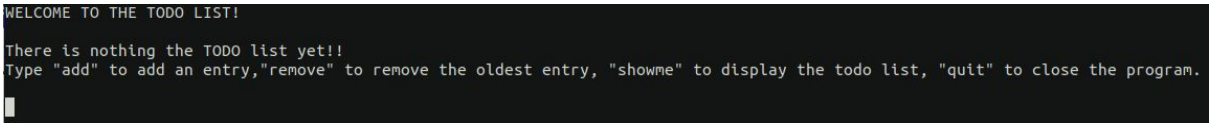
The Object-Oriented implementation is much more uncompromising as objects and data structures are allowed to be used.

## Implementations

### Imperative

The language which I chose to use when developing the Imperative approach was python. This was due to the fact that the python syntax is quite easy to read and understand compared to other languages.

Data is collected from the user by a series of prompts which tells the user what to enter.




To add an entry, the user must specify which type of entry they must use so that the program can determine and present the user with the appropriate input prompts. Adding any entry takes in many input from the user and entries come with many attributes tied to them such as title, date, time and duration. Since data structures are off limits as they are classes themselves in python, all the data gathered from the user inputs are stored in a single string with my queue-like behaviour.

To implement a queue in an imperative way, I used a string as my queue data structure. Adding an entry was simple and included the concatenation of strings to the “queue” string. However, in order to mimic the queue behaviour or *first-in-first-out*, I needed to find out where the task or event entries finish. This was easily done by concatenating  a character divider at the end of every entry. This character could have been anything but I chose the one which is not typically used by many users, the ` (grave accent) character, not to be mistaken with the ‘ (apostrophe) character. This means that I can determine each of the entries in the “queue” which then makes it easy to then remove the first entry by just iterating through the characters on the “queue” string and removing them until the program encounters the divider character ` . Here is an example of what the “queue” string would look like with two entries. (Note the ` divider)

```
<<EVENT>>TITLE: Event 1 DATE: 12/12/2020 TIME: 24:00 LOCATION: Location 1`<<TASK>>TITLE: Task 1 DATE: 21/12/2020 TIME: 24:00 Duration: 5 mins MEMBERS: Jack, John, Joe`
```

I added the feature to display the current status of the todo list as well as the removed/done entries. Utilising another string as a “queue” for the done entries.

```
showme
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~<<TODO>>~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<<TASK>>TITLE: Task 1 DATE: 21/12/2020 TIME: 24:00 Duration: 5 mins MEMBERS: Jack, John, Joe

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~<<DONE>>~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<<EVENT>>TITLE: Event 1 DATE: 12/12/2020 TIME: 24:00 LOCATION: Location 1
```

In the underlying code of this program, functions are used to try to refactor code and to encourage reusability since the program requires many user inputs. I also catered for checking the validity of the dates and times to keep prompting the user for new input when they enter an invalid one.

Overall, the code for the imperative approach is very long for what it’s supposed to achieve and frustratingly inconvenient in terms or storing and manipulating data.

**Object-Oriented**

For the object oriented approach I tried to make the general experience and the user interface the exact same as my imperative implementation. I chose to program in Java programming language due to its embrace in Object-Oriented programming design.

I decided to create a more proper queue data structure using the ArrayList object which would take in data of type “Item”. Removing the 0th element when popping from the “queue” effectively mimicking the *first-in-first-out* behaviour.

I created classes for Task and Event which inherits from the class Item. The “Item” class is a simple one containing what task and Event have in common which are the title, date and time to remove duplication of code. Task and Event have their own constructors which would then cater for their respective requirements, duration & members for tasks and location for events. As well as their own individual toString methods.

Item class

```java
public class Item{
    String title;
    Date date;
    Time time;

    public Item(String title, Date date, Time time){
        this.title = title;
        this.date = date;
        this.time = time;
    }
}
```

Event class

```java
public class Event extends Item{

    String location;


    public Event(String title, String location, Date date, Time time){
        super(title, date, time);
        this.location = location;
    }
```

Task class

```java
public class Task extends Item{

    String duration, members;

    public Task(String title, Date date, Time time, String duration, String members){
        super(title, date, time);
        this.duration = duration;
        this.members = members;
    }
```

I also created my own simple Time and Date classes.

```java
public class Date{
    String day, month, year;
    public Date(String day, String month, String year){
        this.day = day;
        this.month = month;
        this.year = year;
    }

    public String toString(){
        return String.format("%s/%s/%s", day, month, year);
    }
}
```

```java
public class Time{
    String hour, minute;
    public Time(String hour, String minute){
        this.hour = hour;
        this.minute = minute;

    }

    public String toString(){
        return String.format("%s:%s", hour, minute);
    }
}
```

Creating classes is useful for future reusability and modularity of code. Creating objects for task and event entries make them independent from one another thus making it impossible to affect each other's data within their respective instances. Furthermore, each class can contain its class specific methods to manipulate or check its data.

## Summary

In conclusion, the Imperative and Object-Oriented programming paradigms both have their advantages and disadvantages. The Imperative approach is great for learning and understanding each step of the sequence of execution of a program and it is quite easy to get started programming in an imperative style. However, this style becomes quite challenging when you want to scale your program as the complexity and readability of your code becomes more difficult especially when aiming for optimisations.

Object-Oriented oriented approach is one which allows for reusability and modularity of code as well as the added benefits of complex data structures and inheritance to reduce repetition of code. There aren't many cons of programming in this style. Using the object oriented approach might be less time efficient to implement for smaller scale programs as designing classes with optimisation in mind may take longer than other programming paradigms.