Comparing Imperative and Object-Oriented Programming

The goal of this assignment was to implement a calendar program that stores appointments for a one week period in both a imperative and object-oriented programming language.
The languages we chose to do this assignment in where Python 3.6 for the imperative approach and Java for the object-oriented approach. The reason we chose these languages were because firstly in regards to Python, it has a very short and readable syntax and because of this we believed it could best demonstrate the imperative paradigm since if we read it procedurally we can clearly see direct assignment and how the state of the program changes throughout the execution of the program. For our object-oriented approach the reason we chose Java was because everything in Java is an object also because of its statically typed types, these features will greatly help when we come to compare both programming paradigms.

We initially begun with the imperative approach of the calendar program using Python. Before we actually began to implement it a minor bit of planning was required. We firstly had to decide how we were going to represent a calendar in the programme, we quickly decided a dictionary with days (monday - friday) being the keys and an empty list which represented the time in the day as the value would be best mainly due to Pythons dictionaries efficiency when getting a key (O(1)) compared to if we iterated over a list (O(n)).
Once that dictionary was properly implemented and assigned the appropriate name of ‘week’, our next concern was how we were going to implement the following required features, storing an appointment, removing an appointment, displaying appointments for a specific day and showing all appointments for the week with the key feature that appointments could occur at any time within the day and no two appointments could overlap.
For the purpose of readability and to ensure we don’t repeat ourself, we chose to define all of the requirements above as functions.
we initially began by defining a ‘add_appointment()’ function which took a single string as an argument. Our plan was that the user could input the details for an appointment (day, name, start and finish time) as a string and that this function would simply get the correct day from the string check if it is in our ‘week’ and if it was append the rest of the appointment details into the list as a string. This function was inherently very trivial as it just took a string and split all the values based off of a space (‘ ’) and directly assigned them to variables with in the function itself, got the list that was assigned to the day the user chose and appended the rest of the input as a appointment. However with this current implementation many non-logical things could occur. Firstly appointments could overlap and secondly start and finish times could exist outside the range of a day.

To ensure this wouldn't happen we decided to add an additional two helper functions. The first being 'hour2min()' which would take a string that represented a time in 24hour format and convert it to a integer of minutes. The second helper function that was added was that of 'overlap()', which took in the users new appointment they wanted to add and checked if it overlapped with any appointments that were already on that day. If it did it would return 'True' otherwise 'False'. we used the first helper function 'hour2min()' and converted the time so that we could compare already existing times with new incoming times in the 'overlap()' function. We then called the 'overlap()' function from within the 'add_appointment()' so that when the user tried to add an appointment to a specific day it would then check if it would overlap with appointments already in that day by iterating over the list that was assigned to the requested day, parsing the contents of the string which acted as the appointment, retrieving the times of the appointment it was currently iterating over if there were any, converting the times to minutes and checking if either the start time or the end time of the new appointment was lying in between an already existing appointment, if it were it wouldn't append the new appointment and tell the user why, else it would append the appointment to the appropriate days list and let the user know.

The last check we had to do was to ensure that the user entered a valid time in a 24 hour format i.e. (0 < t <24). To do this we simply parsed the users input and ensured that the minutes of the start and end of the appointment was less than 60 (within an hour) and we reuse the 'hour2min()' function and converted the time to minutes and checked it was less than 1440 which is the max number of minutes in a day.

Now our 'add_appointment()' function could successfully change the state of the week dictionary whilst not breaking any rules such as overlapping meetings or impossible times, we could now move onto the final major requirement which was removing an appointment. To achieve this we defined yet another function in our program called 'rem_appointment()' much like the add_appointment where it took in a string but this time all that was required as input was a day and a start time. We chose to use the start time as it would have to be unique unlike the name thanks to our 'overlap()' function and in my personal opinion would be enough for a user if they wanted to remove an appointment.

The 'rem_appointment()' did as follows, checked the time that was inputted was valid the same way 'add_appointment()' done so and enumerated over the specified list, if it found a start time in the list which matched the users input it removed that appointment, changing the state of the program else it did nothing.

The last thing we had to do was to allow the user to input indefinitely until they either typed 'quit' or forced quit the program. This allowed the data to persist.

At the start of this infinite loop we were sure to add a list of instructions. Through the use of if and else statements depending on what the user input was they could now change the state of the calendar by updating and removing appointments. Thanks to the functions we used earlier we were able to keep this core part of the program very clean and readable since all the user was doing was calling on these functions by typing in certain commands for example, 'add' would call 'add_appointment(input())' which would then provide the user with an example and allow them to input their own version. With regards to show day and show week the user would input the correct command 'show'

to display week or ‘showDay’ for a day. These two features were not defined in functions like the other two as they only took a single line to implement and therefore the code maintained good readability. The user could now add, remove, view a day or week of appointments.

Moving onto the object-oriented approach. Much like the imperative approach planning was required however we could inherit and reuse the same concepts from our imperative approach reducing the time it took to prepare and that's exactly what we done. We initially created three classes main, to run the program, Appointment, to encapsulate the fields and methods that an appointment had and Day, which again is encapsulated but also uses composition.
We initially started by implementing an Appointment class which consisted of three private fields name, startTime and finishTime. A constructor that if not passed a valid time will initialize both start and finish to 00:00 and four methods ‘public String getName()’ which returns the name field, ‘public String getStartTime()’ which returns the start time and ‘public String getFinishTime()’ which returns the finish time field. Finally we have our ‘private Boolean isValidTime(String time)’, very similar to our imperative approach, this is called when a new instance of the Appointment class tries to initialize it’s fields, specifically the time fields. It parses the string and ensures that the hours is less than 24 and that minutes are less than 60. If they are the object is initialized with the parameters that were passed into its constructor else the times get initialized to 00:00 as stated previously.
Once we were happy with the Appointment class and that it encompassed all the relative information in it we began to write out the Day class.
The Day class consisted of solely a name which would be the name of the day and a List Array of type Appointment to act as the time in that day. By having our List Array of type Appointment this means we cannot have anything that is not an Appointment and since our appointment class that we previously defined encapsulated all the necessary data required this means that every occurance of a our List Array of ‘appointments’ will also have these exact same well defined traits.
We only added the name field to the constructor since different instances of day will have different names but all appointments in days will start out the same, empty.
We then define six methods in total. ‘Public void append(Appointment app)’, this will take an appointment and firstly check if the time is set to 00:00 for both start and finish. If this is so then we tell the user what they inputted is invalid, we also check if times overlap, ‘private Boolean overlap(Appointment app)’, firstly the method is private because it does not need to be called outside this class, doing this can increase security especially with fields. We take in the same Appointment that was passed to the ‘append ()’ method get the start and finish times and again convert them to minutes similar to the imperative approach, ‘private int timeToMinutes(String time)’, this takes the string that consists of a time and parses it into hours and minutes. We then have to convert the type from integer to string to perform the multiplication and addition needed. Once the types have been converted the hours multiplied and the minutes added, we return our newly calculated integer back to our ‘overlap()’ method where we iterate over our Array List of Appointment called appointments and check to see if our new times lie between

any currently available appointments if so we return true to say that it does indeed overlap else we'll return false. We then return to our initial method 'append()' and if there is a overlap we output to the user that there is one, else we continue down and add our new Appointment to our Array List of Appointment. As we have seen, our Day class has a 'has a' relationship with our Appointment class where it has many appointments. This is a key feature of object-oriented programming called composition. The next method in our Day class is that of 'public void remove(string startTime)'. This much like our overlap method before in regards to iteration will iterate over our Array List of Appointment and compare a string of start time to all the appointments in appointments. If there is a match we remove the Appointment, notify the user and break out of the method. The final two methods that we haven't mentioned are the getters of our class which just return our fields. If the field is private this still allows us the programmer to get the value from the field while not directly interacting with it.
Our final class is that of main, it is here where we call and create new instances of our two previously defined classes. This class very much resembles that of the imperative approach where we defined our 'week' in this case it is a LinkedHashMap, I chose to use this rather than a HashMap since it maintains order. We created several new instances of the day class and passed in each day of the weeks suffix ('mon') into its constructor. From here on forward we start a infinite loop of user input. If the user enters 'add' and return, then they enter the required parameters, an example is outputted to the users prompt, they correctly input that example or their own variation then what occurs is the input gets split, we search our LinkedHashMap for the day they entered and then we call 'append()' method in the day class and we pass it a new instance of the Appointment class which gets initialized by the users input, this will create a new Appointment object, check the times that were passed into its constructor are 'isValidTime()' it then goes to the Day class and does all the checking it has to do their such as is the time given in the scope of a day or is their a 'overlap()'. If everything succeeds then our new Appointment object will successfully be added to our list array of type Appointment called appointments in our specific instance of Day which we initialized in our main class. If we wish to view a day or week all the user would input is 'showday', return, the suffix of the day they wish to view and for week they'd just enter 'show'. This basically works the same way as the imperative approach. Much like how our 'add' input calls the Days 'append()' method , the user can input 'remove', return, and follow the example to remove an appointment of their choice where based of the users input the program could locate the correct instance of the Day class that contains the soon to be removed 'Appointment()'.

Once both calendar implementations were complete it is very easy to see the key differences between the two programming paradigms. Initially when programming the Python approach all of our code was centralized in a single file, this was fantastic for understanding how the program worked since you could easily read the code procedurally as if it were a book with the exceptions of functions. This was not the case with the Java implementation since it was scattered across multiple files, to be exact three, having separate files can be a bit more tedious to search through and read however in large scale programs this can prove extremely beneficial because rather

than read through a single lump of code which occurs with imperative programming you can find code easier with a object oriented approach if the classes follow a good naming convention, since only methods and fields relative to a class will be places in that class. On that note you can implement multiple classes in a single file in Java however it is bad practice therefore would be rarely seen.

Whilst writing out both implementations of the calendar program I found that while doing the imperative approach it was much easier to implement code without proper planning since all you were doing was altering, updating and changing the state of the program, in our case the state would be the dictionary that we defined that held all the relevant data we cared about. However in Java even though more planning was required in order to specify what classes were needed and what fields and methods they should have we were essentially changing the state of our LinkedHashMap which also acted as our week.

One key feature that I saw stand out among both paradigms was robustness and security. During the Python implementation I could directly assign variables and overwrite other assignments easily due to its imperative nature but with Java like we mentioned once or twice throughout the process of implementing of the calendar app, both of the classes we created made use of encapsulation. By using this we could completely remove the feature to be able to directly interact with variables and stop direct assignment. We could also make use of abstraction by implementing either a interface or abstract class, however I did not do this as the program was too small and didn't require security as a feature. Also with regards to the Appointment class thanks to its strict fields, in order to create a Appointment the user had to enter those exact three fields all the way down to the type, if this did not happen the object wouldn't be instantiated however we did not have these kind of strict limitations in Python instead we allowed the user to input a string and hoped they could follow and stick to the format that was required and it was only once the data was inputted and saved was it then checked.

Also in our imperative approach we were stuck to using default data structures and types such as strings and ints but with the latter approach we explicitly saw thanks to Java's static typing that our classes could now be seen as types, we especially saw this with the composition of the Appointment List Array, this can be fantastic for creating robust programs since it can display errors that would normally go unnoticed.

Finally one major difference that exists between both object-oriented and imperative paradigms is that of scalability. If the assignment specification were to change right now and increase the calendar from a single week to a month the imperative approach would have a difficult time quickly scaling up to that requirement compared to the object-oriented approach since we would have to rewrite the majority of the core logic however with the object-oriented approach we could simply create a Week class much like the day class that could inherit our Day class and nicely encapsulate all the necessary requirements without having to change any of the other classes with the exception of the main class however these changes would be minor.

This concludes both the imperative and object-oriented implementations of the calendar program and the comparison of both programming paradigms.