# MEMORY

OOP PRACTICE 1 - JSONTMB

Login: arcadia.youlten and felipe.perez

Arcadia Youlten and Felipe Perez

## Contents

## Summary of the Statement

In this practice, the TMB’s transportation web application was somehow rendered inactive, and the task given to us was to fix it. The app functions included storing user information, allowing users to create locations and routes, allowing users to set their favorite locations and stops, showing the user their location and route history, showings users stations inaugurated on their birth years, allowing users to search locations, and showing users bus wait times depending on what stop code they enter. The goal of this practice was to learn how to build urls and learn how to work with a given API by working with the TMB web app.

When the user entered the program, they were asked for their name, email, and birthyear. After this, the user had several options: to search locations, plan a route, or look for the bus wait time. When the user searches for locations, they are asked for the name of the location they would like to look for. If the name matches any previous location saved by the system, then its information is displayed. If the user wants to plan a route, then they are asked to input all relevant information (origin, destination, day, time, departure/arrival, maximum walking distance), and if the TMB can make a route between the two given points, the route is shown and stored. If the user wants to find the bus wait time, then they are asked to input a stop code. Once the stop code is input, the buses arriving to that station are displayed.

However, if the user chooses the option user management, another menu is shown. This menu has more options, namely, user locations, location history, user routes, user favorite stops and stations, and stations inaugurated by birth year. User locations displays all of the locations that the user has created so far, or displays a message if the user has not created any locations. The user is always asked if they would like to make another location. Location history shows all of the locations the user has previously searched for in menu 1. User routes displays all of the routes the user has previously searched for in option 3 of the main menu. User favorite stops shows all of the stops and metro stations within 500m of a user’s favorite location. Stations inaugurated by birth year displays the stations inaugurated in the user’s birth year.

## Design

In total, we had 23 classes. Initially, we had fewer classes than this, however, as we continued to program, we not only figured out how to store the information we needed appropriately. We also had 5 packages, which allowed us to organize our classes in a coherent manner.

Main has a unidirectional associative relationship with both menu and logic. This is because main contains an instance of logic and menu. Logic and menu have a unidirectional relationship, as logic has an instance of menu, and uses menu's functions. All of the exceptions have a unidirectional relationship with logic. This makes sense as exceptions are only processed in logic, so logic contains the instances of the exceptions. Java.lang.exception is inherited by all of the exception classes, this allows us to use the throw method on the classes which we need them. User and Location have a bidirectional associative relationship. This is because user needs to keep track of which locations they've created, and which locations they've searched for. From user and location's relationship, the dependency FavLocation is generated. Restaurant, monument, hotel, and place inherit from Location, which makes sense, as these classes should be considered locations. Therefore, they should share the same attributes. Logic has a unidirectional associative relationship with busStation, API, and metroStation, and JsonParser. Logic has instances of busStation and metroStation so that it can check which stations exist depending on what happens. JSONParser has a unidirectional associative relationship with generic, as the returned information from the JSONParser is an arraylist of Generic objects. Logic has a dependency with iBus and Section. Walk and Transit inherit from Section. No matter what part of the journey they're on, a user must either be walking, or on some form of transit. iBus, Route, busStop, and metroStation all have a dependency with API. The information returned from the API changes depending on what the user has requested, therefore, these classes that store information from the API must be dependent on it. Generic has a dependency with restaurant, hotel, place, and monument. This is due to the fact that the information in any location object depends on the information parsed from the generic array.

# Class design and Extras

## DataModel

The DataModel package contains all the classes related to the information contained in the file 'Localization'.

### FavLocation

FavLocation is a class that stores information about a location that the user has chosen to mark as their favorite. It has the following attributes: location, date, and type. Location is a Location object that can be one of the following objects: a restaurant, a monument, a hotel, or a place. The date is of type Date which stores when the user set it as their favorite location, and the Type is a string of one of five options, home, work, studies, leisure, culture.

FavLocations has four methods, three of which are getters. The other is a constructor which is passed all of the attributes mentioned above.

### Generic

Generic is an auxiliary class that enables us to retrieve information from the Json file effectively. It is used to store and access information about all types of locations before officially being defined as a class. If we did not use this class, it would be impossible to effectively get all the information from the JSON file, and we would lose important details.

The attributes of Generic include name, coordinates, description, stars, characteristics, architect, inauguration. Name is a String that contains the location name, coordinates is an array of doubles that contains the coordinates of the location in the EPSG: 4326 format. Description is a String that describes information about the location. Since all locations in the JSON file have this information, it makes sense to include these attributes in Generic. Stars is an Integer which represents how many stars a hotel has. Architect is a String which contains the architect name, and the Integer inauguration is the date the monument was inaugurated. Characteristics is a JSON array of characteristics representing a restaurant.

The methods of Generic include getters for each attribute. Since we do not change any of the information of existing locations, it does not make sense to include setters in this instance. Additionally, Generic is an auxiliary class, so using setters would not make sense.

### Hotel

Hotel is a class that extends from location, and it is a location that not only includes all attributes of a location, but also stars, an Integer representing how many stars the hotel has.

The methods of hotel include a constructor that takes a Generic object, and a setter and getter for the stars. The constructor with the generic object is used to get the information from the generic object, and then afterwards, the set stars will add the star value to the object.

## Location

Location is an abstract class that represents the different places that can be searched for or stored using the app. It is an abstract class as we can have a few different types of locations, namely, hotel, monument, and restaurant. Additionally, some locations do not have any unique traits, but, because location is an abstract class, we cannot directly make a new location object. Therefore, we created a class called place, which functions as a generic location object.

The attributes of location include name, coordinates, and description. These are the three generic attributes that each location has. Name is a String that contains the location name, coordinates is an array of doubles that contains the coordinates of the location in the EPSG: 4326 format. Description is a String that describes information about the location.

The methods of location include three getters and two constructor methods. One of the constructor methods creates a location given a Generic object, and the other creates a location given the three attributes mentioned above.

## Monument

Monument is a class that extends from location; therefore, they share all the same attributes. However, monument also contains two extra attributes: Inauguration, and Architect. Inauguration is an Integer representing when the monument was inaugurated, and Architect is a String that represents the name of the architect who constructed the monument.

The methods of monument function similarly to those of Architect, where there is a constructor that uses a generic object to get part of the location's information. However, there are two getters and setters, one for architect, and one for inauguration.

## Place

Place is a class that extends from location, however, there are no other attributes that make this class unique. Since location is an abstract class, we cannot directly create a location object that can be stored in an array. Instead, we use an object of this class type. Since this class is supposed to be the default location, it has the same attributes and methods as locations.

## Restaurant

Restaurant is a class that extends from location; therefore, they share all the same attributes. However, restaurant contains an extra attribute, Characteristics.  Characteristics is a JSON array representing the characteristics of the restaurant.

The methods of Restaurant function similarly to those of Architect, where there is a constructor that uses a generic object to get part of the location's information. However, there is an extra getter and setter for characteristics.

### User

User is a class that stores all of the relevant information about the current user of the program. The user class has the following attributes; username, email, birthyear, userLocations, favoriteLocations, pastRoutes, and searchedLocations. Username, email, and birthyear are all attributes input by the user at the start of the program. The most important attribute out of this group is birthyear, as we use this attribute to determine if there were any stations inaugurated in the user's birthyear. UserLocations, favoriteLocations, pastRoutes, and searchedLocations are all arraylists of different types. UserLocations stores the locations the user has created, favoriteLocations stores the user's favorite locations, pastRoutes stores the routes the user has searched for in the past, and searchedLocations has stored the locations that the user has searched for in the past. Since all of this information is unique to each user, it makes sense that we would store it in user, rather than another class with a similar function like logic.

Despite having so many attributes, the user class only has four methods, three of which are getters. Since we only set the user information when the user enters the program, we created a constructor that takes these attributes and sets them all at once. The getters are for the three attributes, username, birthyear, and email.

## Exceptions

The 'Exceptions' package contains classes that extend from java.lang.exception. These exceptions are created to simplify which error messages should be displayed when. None of the exceptions have attributes.

### LocationNotFoundException

The method printErrorMessage prints out an error message saying "Sorry, there is no location with this name" when called.

### StationNotFoundByYearException

The method printErrorMessage prints out an error message saying "No subway station opened your birth year :(" when called

### StopCodeInvalidException

The method printErrorMessage prints out an error message saying "Error, stop code not valid!" when called

## Parsing

The parsing package contains all of the methods and attributes necessary to parse a given file.

### JSONParser

JSONParser is a class that implements the interface method parseLocations. It has no attributes. In this case, parseLocations takes the JSON file "localizations.json" and gets the arraylist of locations from the jsonObject "locations" contained in the file. In order to successfully store all of the locations with their correct information, we create an arraylist of Generic class objects, so they can be sorted and put each location into their appropriate category. To do this, we use a for loop that runs for the entire size of the Genneric arraylist, and check which attributes of each location are stored. For example, if a location has stars, then we create a Hotel object, and use the constructors and setters to put that location's information into the Hotel object. Then, that object would be added to an arraylist of all locations of type locations. If the file cannot be read, then an error is displayed and the program is exited.

### Parse

Parse is an interface which contains the method parseLocations(). This method returns an arrayList of locations, as after using this method, all the location information should be stored correctly in its array. Parse is an interface because we could potentially want to change the way we parse the information or change the file type of what's being parsed. For example, if the developer were given a CSV file that contained all of the locations instead of JSON file, the only thing that would need to be coded would be a class that uses the parseLocations() method, with the actual parsing part coded in a different manner.

## System

The system package contains all classes that relate to processing user information, or displaying information related to user information.

### Logic

The logic class contains all information related to processing all user input. The attributes of logic are minlong, maxlong, minlat, maxlat, user, api, parser, scanner, allLocations, busStops, and metroStations. minlong, maxlong, minlat,  and maxlat are constants of the minimum and maximum values for longitude and latitude. These are used to check if the coordinates the user inputs when creating a new location are within an acceptable range. Since logic is where all of the functions are called, instances of user, api, parser, and scanner are needed. AllLocations is used to store information about the locations of the system, including ones that the user creates. busStops and metroStations are arraylists used to store information about busLines and metroStations that the TMB controls.

The methods in logic are: loadData, Intro, listLocations, userCreateLocation, createNewLocation, validLocationName, askForLongitude, checkIfLongitudeCorrect, askForLatitude, checkIfLatitudeCorrect, searchLocation, checkLocationExist, userSetFaveLocation, locationHistory, findStationsByYear, stationsInauguratedByBirthYear, getBusWaitTime, checkStopCodeIfExists, checkIfStopIsFavorite, showCloseStations, planRoute, userRoutes, calculateTimeInMinutes, checkYesOrNo, distanceInMBetweenEarthCoordinates, degreesToRadians, whichOptionM1, whichOptionM2.

LoadData loads the location information from the JSON file, and loads the bus stops and the metro stations from the API. It returns a boolean, true if the data is loaded correctly. Other methods from other classes are called within this method.

Intro asks the user for their username, email, and birthyear. From there, the user constructor is called and a new user is generated.

ListLocations lists the locations the user has created. If no locations are created, a unique message is displayed. Additionally, the user is asked if they want to create a new location. If yes, userCreateLocation is called.

UserCreateLocation asks the user for different parameters about their location they want to create, and creates a new location based off of this information. If the user inputs a name that already is allLocations, an error is displayed.

CreateNewLocation calls the Place constructor and passes all user information to create a new place.

ValidLocationName checks if the name the user has input when creating a new location already exists in the allLocations array. The checking is done via equalsIgnoreCase and a for loop. It returns a boolean, when true, the name already exists in the file.

AskForLongitude asks the user for the longitude of a location. If the input is not a double, then the input will not be accepted.

CheckIfLongitudeCorrect checks if the longitude of the location is between minlong and max long

AskForLatitude asks the user for the latitude of a location. If the input is not a double, then the input will not be accepted.

CheckIfLatitudeCorrect checks if the latitude of the location is between minlat and maxlat

SearchLocation asks the user for the name of the location they would like to search for, and then displays information about that location if the information is found.

CheckLocationExist checks if a location exists in the allLocations arraylist. If the location is not found, then the locationNotFoundException is thrown. It returns an int which is the position of the found location in the arraylist. If the position is not found, the value is set to –1.

UserSetFaveLocation allows the user to set a location as favorite. The location must have a type of home, work, studies, leisure, culture. The date is generated and stored in the fave location object, and the favLocation constructor is called.

LocationHistory displays all of the locations the user has searched for previously. If there are no locations that have been searched for, a message is shown. It works via a for-loop that gets the name from user.searchedLocations.

FindStationsByYear finds all of the stations that have the same inauguration year as the user's birth year. This is done by comparing the first four characters (which represent the year) of the inauguration date and the birth year converted to a string. If no locations have been added to the arraylist birthyear_stations, then stationNotFoundByYearException is found

StationsInauguratedByBirthYear print out all of the stations inaugurated in the user's birthyear. If there are none, a unique message is displayed.

GetBusWaitTime asks the user for a stop code and displays the wait time until the next bus(es) come to the station. If the station does not exist, or the user does not put in an int when asked for the stopcode, then the user will be notified.

CheckStopCodeIfExists checks if the stop code entered by the user is a stop code that corresponds to a given station. If one is not found, the stopCodeInvalidException is thrown.

CheckIfStopIsFavorite checks if a stop is a favorite when given an arraycode and the list of bus stops from the API. It also checks if that stop is close to a favorite location.

ShowCloseStations shows the stations and stops close to the favorite locations the user has set. They are displayed if within 500m of the station. If the user has not chosen any favorite locations, a message is printed.

PlanRoute asks the user to input all the necessary information to find a route between two points (origin, destination, departure/arrival, day, time, and max walking distance). Once the information is processed from the API, and everything is correct, then the route is saved to a list of searchedRoutes by the user.

UserRoutes prints all the routes the user has saved. If there are none, a unique message is printed.

CalculateTimeInMinutes calculates time for a section to be completed. Since the start time and end time of a section are given in Epoch time, we can simply convert from miliseconds to seconds to find the time. Returns the amount of time it takes for a section to be completed.

CheckYesOrNo checks if the string input is a yes or no. Returns a boolean, true if the answer is yes or no.

DistanceInMBetweenEarthCoordinates given two sets of coordinates, converts the distance between them to meters. Returns a double, the distance between two coordinates in meters.

DegreesToRadians converts degrees to radians. Returns a double, the degrees converted to radians.

WhichOptionM1 chooses a method to run depending on what the user has input on the first menu.

whichOptionM2 chooses a method to run depending on what the user has input on the second menu.

## Menu

The menu class contains all information related to displaying information the user inputs. The menu attributes are mostly, if not all, constant Strings or ints. This is because the menu stores the information that the user should see on the screen when the menu class is called. The menu has the following attributes: MIN1,MAX1, select, intro, userinfo1, userinfo2, userinfo3, flagvalid, option1menu1, option2menu1, option3menu1, option4menu1, optionamenu2, optionbmenu2, optioncmenu2, optiondmenu2, optionemenu2, exitmenu2, exitmenu1, option1, option2, scanner. MIN1 and MAX1 are int values that represent the minimum and maixmum values that the user can input on the first menu. Select is a String which says "Select an option". Intro, userinfo1, userinfo2, and userinfo3 are all strings that are displayed at the start of the program when the user is asked for their information. Flagvalid displays when the user has input all of their information correctly. All of the optionXmenuN attributes represent the different strings which have the text of the menu options in them. Option1 and option2 represent the information that the user inputs when on the menu. Scanner is a class that takes user information from the keyboard.

The methods in menu are a constructor, printMenu1, printMenu2, exitMenu1, askForOption, askForOptionString, validOption1, and validOption2.

The constructor creates a new menu object with a scanner and changes the default value for option 1. the default value is of option is initialized to a value other than the 5 possible choices for the user, so that one of those functions is not automatically called when calling the menu class.

PrintMenu1 prints the different options available for menu 1

PrintMenu2 Prints the different options available for menu 2

ExitMenu1 checks if the user has input the value that exits said menu. It returns a boolean, if the user has entered the exit value, the menu returns true.

AskForOption asks for an int option for menu 1. If the user inputs a value that is not an int, then the value for option1 is set to 0.

AskForOptionString asks for an string option (for menu 2).

validOption1 checks if the option entered is valid for menu 1. It returns a boolean, true if the int entered is within the range of the menu.

ValidOption2 checks if the option entered is valid for menu 2. It returns a boolean, true if the string entered matches one of the menu options.

## WebServices

The webservices package contains all of the information that relates to any calls to the API.

### API

The API class contains all of the methods which access the API in someway. API only has one attribute, client. Client is a private OkHttpClient which allows us to make requests to websites using urls.

The methods in API include loadBusStops, loadMetroStations, getBusWaitTime, and plannerAPI. LoadBusStops loads the bus stops from the API call to the TMB webservice. If something goes wrong, then an error message will show, and the arraylist storing the values will be set to null. It returns an arrayList of busStops.

LoadMetroStations loads the metro stations from the API call to the TMB webservice. If something goes wrong, then an error message will show, and the arraylist storing the values will be set to null. It returns an arrayList of metroStations.

GetBusWaitTime gets the wait time for the next bus(es) at a stop the user requests. It returns an arraylist of buses that arrive to that stop. If no buses arrive, it is returned with nothing inside. The requested URL is built from the user's input stopcode.

PlannerAPI requests route info from the web after the appropriate url has been built from the user inputs (origin, destination, date, time, departure/arrival, max walk distance). It returns a route, which displays on screen if sucessfully constructed. If something goes wrong (either with the request, or the user input) the route is returned as null, and an error from the webservice is printed.

### BusStop

The class BusStop represents a bus stop. We do not store all of the attributes from the webservice, but rather only those we need. The attributes of busStop include coordinates, stop_code, stop_name, and stop_descripton. Coordinates are the coordinates of the bus stop. Stop_code is the stop code of the bus stop, stop_name is the name of the stop, and stop_description is the description of the stop. We need stop_code, and stop_name for use in finding routes and additionally for use in bus wait times.

The only methods are a constructor and getters. The constructor makes a busStop object given a JsonObject with the correct attributes.

### iBus

The class iBus represents a bus line that circulates through a particular stop, including how long it will take to reach that stop. The attributes include, route_id, line, time-in_text, time_in_sec, destination, time_in_min. The route_id is an integer which identifies the route, and line is a string that identifies which line stops at that station. Destination identifies the destination of the route. Time-in-sec represents how many seconds until the next bus arrives at the station, while time_in_min represents how many minutes until the next bus arrives at the station.

The only methods are a constructor and getters. The constructor makes an iBus object given a JsonObject with the correct attributes.

### MetroStation

The class BusStop represents a bus stop. We do not store all of the attributes from the webservice, but rather only those we need. The attributes of busStop include coordinates, station_id, station_code, station_name, line_name, and date_inagurated. Coordinates are the coordinates of the bus stop. Station_id and station_code are ints that identify the stop they are assigned to. Station_name is the station name, the line_name is the line name, and date_inaugurated is the date the station opened.

The only methods are a constructor and getters. The constructor makes a metroStation object given a JsonObject with the correct attributes.

### Route

Route represents a route that the user can plan or see. Route has the attributes origin, destination, date, time, maxWalkingDistance, timeTaken, and routeSections. Origin and destination are the coordinates of the origin and destination of the route. MaxWalkingDistance is the maximum walking distance the user wanted to walk on the route. Time taken is the amount of time it took to follow the routes, and route sections contain information about each leg of the journey

The only methods are a constructor and getters. The constructor makes a metroStation object given an arrayList of sections, a date, time, maxWalkDistance, origin, destination and timeTaken.

### Section

Section is an abstract class that represents a certain step of the route the user wants to take. It is extended by Transit and Walk. The attributes of Section are start_time, end_time, and mode. Start_time and end_time are Epoch time measures of the time that the user starts a certain section, and ends that section. Mode indicates how the user is being transported, walking, via bus, via transit, or another descriptor.

### Transit

Transit represents a section where the user takes some form of transport (bus, metro, rail, ect.). Transit extends section. The attributes of transit include line_name, from_stopcode, to_stopcode, from_name, to_name. Because the user will go from one location to another on a particular line, we need both stopcodes and names.

The only methods are a constructor and getters. The constructor makes a transit object given a section jsonObject.

### Walk

Walk represents a section where the user must walk. It only has a constructor method as it does not have any other data needed other than that given in section.

## Main

Main does not contain any methods or attributes. This is because we used it as an area that could simply call functions.

## Testing methods

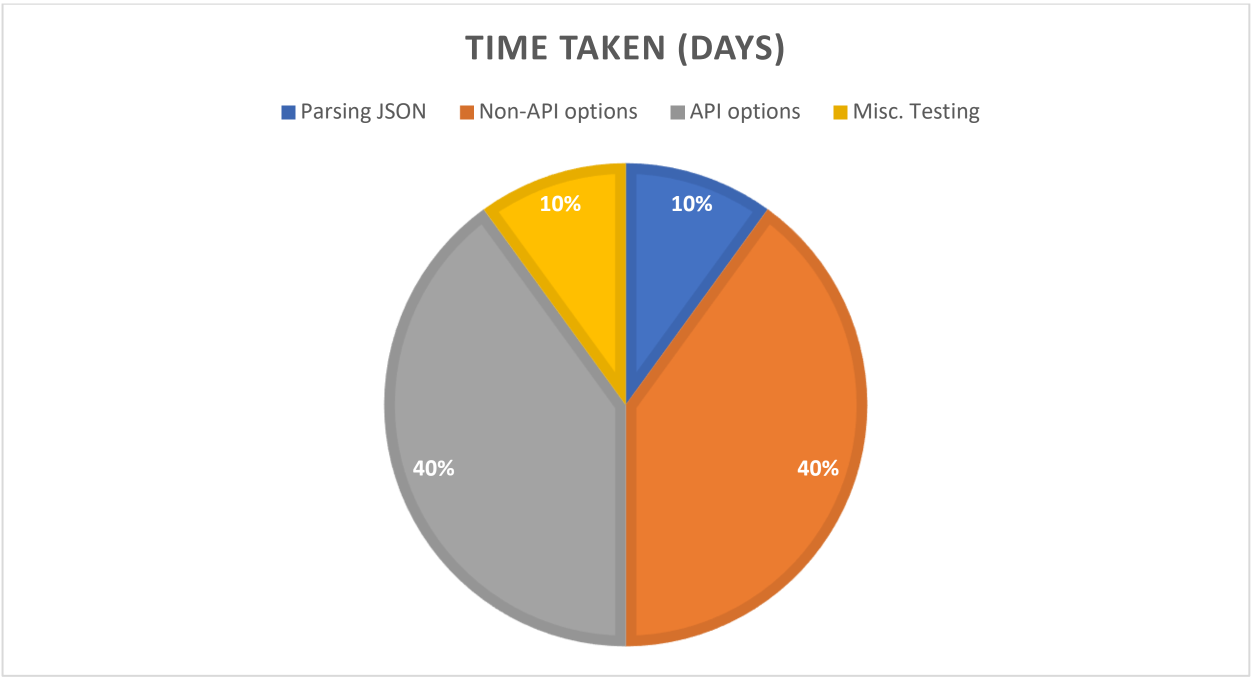
In order to test our code, we used a combination of methods. To check if information was being taken by the scanner correctly, we input the different types of information that would cause an undesired result. For example, when testing our conditions for the scanner to accept an integer, we would just input a string instead. If our criteria were not met, the program would crash, and we would go back to try and fix the problem.

For testing the API, we first tried to figure out how the URL was supposed to be encoded. This was more or less straight-forward, as we simply had to look at the documentation to determine what needed to be used. After we found how to build the URL, we would create a sample one that had valid values that we could test. Using the sample URL, we printed the different kind of information contained in the JSON object. By doing this, and by manually looking at the JSON object we received, we were able to determine which pieces of information we needed to store, and the others which we could ignore. After processing the information, and ensuring that we could make it display correctly, we started building the URL from the user inputs. We also tested the results when we deliberately input wrong information, resulting in a different kind of JSON object being returned. We used this new object to print out errors when needed.

For testing the JSON, there was a lot of trial and error. Most of the problems stemmed from not understanding that the JSON contained a JSON object which was a JSON array. However, once this problem was solved, getting and storing the information was quite straightforward. By creating a class that was extended by all location types, we were able to create an arrayList of objects that fell under the ‘location’ category. From this point, it was simply a matter of retrieving the information necessary in the object.

For testing other functions and classes we more or less used print functions in order to check what was happening. Other than that, we forced variables to have certain values depending on what we wanted to test.

## Dedication



In total, this practice took us roughly 5 days of complete programming to complete. The options that took us the most time were option d and option 4. These were somewhat more tedious than the other options as we were not quite sure how to work with the API. Once we knew what the API request returned, we were able to effectively glean the necessary information to use from the returned JSON file. Additionally, the JSON file seemed difficult to parse at first, but after understanding its structure it was more or less straightforward. Figuring out how to store the information from the JSON file took some time but was quickly overcome with some testing. Since we took the work step-by-step, programming this project was quite straight-forward, and there were no particular points where we got too hung up on how to implement an idea. Instead, we were spending our time doing the implementation ourselves or figuring out ways to improve it.

There was nothing about this practice that was particularly difficult, however, we would attribute this more to the good organization on our part. Our initial class diagrams were quite messy, but after restructuring and reorganizing, we had a decent design to work with. This massively helped in seeing which pieces of information should go where, and how many instances of certain objects we should have. For example, we decided that it would not make sense to have too many user objects when running the program for the first time. We included this in our class diagram, and from there we tried to organize our program around this idea. In other cases however, we strayed from the class diagram. For example, when implementing the Route class, we did not realize that we would need classes for the different sections of the journey. Instead of trying to only use Route to work with the sections, we instead added more classes. This allowed us to stay organized and helped us understand our code better. We believe it is this mix of flexibility and organization that allowed us to successfully complete this project

## Conclusions

In conclusion, this project was useful to learn how to improve our skills with Java. In another class, we had to turn in a different practice which required us to use Java. When we completed that practice, we felt that we did not make a strong enough attempt to organize our classes and we felt that we did not use of all Java's capability as an object-oriented language. However, in this practice, we made more effort to reach those goals. On several occasions, we stopped programming and thought about a more efficient or more object-oriented implementation of our code. Not only did this save us time later by eliminating reorganization time, but we were able to complete the practice in a reasonable amount of time.

The hardest part of this project was learning to work with the API. Although not conceptually difficult, we had never worked with an API before, and so we were unsure of how it was supposed to work. Doing various tests and looking at what the different URLS returned, we were able to effectively pick which pieces of information that we needed to store, and those we did not. After organizing and testing, this section became much easier.

If we were to redo this project, we would probably be able to more efficiently work with the API and the JSON file. Originally, we had made an extra class to process the JSON object received from the file, however, we realized that we did not need it.

## Bibliography

Baeldung. “Introduction to JavaDoc.” *Baeldung*, 7 May 2019, www.baeldung.com/javadoc.

“Date.” *Date (Java Platform SE 8 )*, 11 Sept. 2019, docs.oracle.com/javase/8/docs/api/java/util/Date.html.

“DYclassroom.” *Brand*, www.dyclassroom.com/java/java-returning-objects-from-methods.

“Java Platform, Standard Edition Java API Reference.” *Java.lang (Java SE 13 & JDK 13 )*, 13 Sept. 2019, docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/package-summary.html.

“Javadoc All Exceptions.” *Java Practices->Javadoc All Exceptions*, Hirondelle Systems, 2018, www.javapractices.com/topic/TopicAction.do?Id=44.

Peitek, Norman. “Gson - Mapping of Nested Objects.” *Future Studio*, futurestud.io/tutorials/gson-mapping-of-nested-objects.

Square, Inc. “OkHttp.” *OkHttp*, 2019, square.github.io/okhttp/.