**Final Project**

**README**

**Overview**

This program will take in two different tsv files, one containing the titles of the train stops along with the information for each stop, and one containing the links between the stops. The 15 train stops are inspired by the city of Chicago and there are three types of trains that travel 24/7 in our program: Express, Tour, and Normal. The express train stops at the following stations: O’Hare, The Loop, Chicago, Midway, U Of Chicago, Water Way, and South Side Business. The tour train stops at the following stations: O’Hare, Jefferson Park, United Center, Belmont, Wrigley Field, Soldier Field, Water Way, White Sox, and Juice WRLD Memorial. The normal train will stop at each stop available in our program. The program is able to take in which type of train that the user would prefer to go on, if any, along with the day and time of departure. The program will then give the user information on the route from the desired start stop to the desired end stop, such as whether there is a bathroom, the chances the bathroom is taken, the estimated cost of the trip, the chance of boarding the train, the chance of getting a seat, and the estimated time of arrival at the final destination. Furthermore, if the user would like food on the train, there is also information on the menu for that day and the cost depending on the type of train. Finally, the program will give the user more information on their desired final stop, such as what type of station it is, if the stop has a bathroom, and what type of food is directly around the stop.

**Usage**

To use the program, run the following code in the terminal, with stops.tsv file going first then the stopslinks.tsv file going next. Here is an example below:

% javac \*.java

% java MetroNavigator stops.tsv stopslinks.tsv

The program will welcome the user and ask if there is a preference for the type of train for travel; if not, the user can just type “All”. Here is an example of what that would look like:

Hello! Welcome to the Metro Navigator!

Please select the type of train to check your travel:

(Enter Express, Tour, Normal or All)

If the user selects “Express”, the program will prompt the user to enter in their start stop, then their end stop. Since O’Hare and Water Way are two stations that the express train stops at, we will use O’Hare as our example start stop and Water Way as our example end stop:

Please select the type of train to check your travel:

(Enter Express, Tour, Normal or All)

Express

Start Stop:

O'Hare

End Stop:

Water\_Way

As one can see, the user must enter an underscore if there is a space in the station name. If the user enters a station that the express train does not stop at, such as Juice WRLD Memorial, the following error statement will print out and prompt the user to enter in another station:

Start Stop:

O'Hare

End Stop:

Juice\_WRLD\_Memorial

Express train does not stop at this stop.

Please pick a stop that the Express train stops at.

After entering in the start and end stop, the program will ask the user to enter in a time and day of departure to give the user the most accurate information. Once the user enters in this data, the program will print out information related to the desired route of the user. We will use 6:30 AM on a Friday as our example:

Enter your time of departure in a 24-hour format.

Ex) 1:52PM -> 13:52, 6:30AM -> 06:30

06:30

Enter your day of departure. (One day)

Ex) Monday, Friday, Saturday...

Friday

\*\* Here is your route from O'Hare to Water\_Way on Express train. \*\*

[ O'Hare -> The\_Loop -> Chicago -> Midway -> U\_Of\_Chicago -> Water\_Way ]

Travel Summary (Express Train):

Express Train runs every 30 minutes.

Has Bathroom?: No

Estimated Cost: $8.0

Chance of Boarding: 43%

Chance of Getting a Seat: 26%

Duration of Travel: 66 minutes

Estimated Time of Arrival: 07:36

Here is some information about your end stop.

Name of Stop: Water\_Way

Station Type: Park

Has Bathroom?: Yes

Food at Stop: Fancy Food

If the user selects “Tour” or “Normal” as the type of train, the program will print out information in the same format (tour and normal trains will also have food on the train, which will be shown in the following example), as long as the type of trains stop at the stations that the user enters. However, when a user selects “All” as the type of train, the program will print out information about the desired route for each type of train, which truly gives the user a chance to decide which train would be the best to take, in terms of comfort and/or time. Let us use O’Hare and Water Way once again as an example as all three types of trains stop at those two stops:

Please select the type of train to check your travel:

(Enter Express, Tour, Normal or All)

All

Start Stop:

O'Hare

End Stop:

Water\_way

Enter your time of departure in a 24-hour format.

Ex) 1:52PM -> 13:52, 6:30AM -> 06:30

06:30

Enter your day of departure. (One day)

Ex) Monday, Friday, Saturday...

Friday

\*\* Here is your route from O'Hare to Water\_Way on Express train. \*\*

[ O'Hare -> The\_Loop -> Chicago -> Midway -> U\_Of\_Chicago -> Water\_Way ]

Travel Summary (Express Train):

Express Train runs every 30 minutes.

Has Bathroom?: No

Estimated Cost: $8.0

Chance of Boarding: 57%

Chance of Getting a Seat: 19%

Duration of Travel: 66 minutes

Estimated Time of Arrival: 07:36

\*\* Here is your route from O'Hare to Water\_Way on Tour train. \*\*

[ O'Hare-> Jefferson\_Park-> United\_Center-> Belmont-> Wrigley\_Field-> Soldier\_Field-> Water\_Way ]

Travel Summary (Tour Train):

Tour Train runs every 45 minutes.

Has Bathroom?: Yes

Chance of Bathroom being full: 17%

Estimated Cost: $10.0

Chance of Boarding: 90%

Chance of Getting a Seat: 3%

Duration of Travel: 69 minutes

Estimated Time of Arrival: 07:39

Menu for food on Tour Train:

Burger: $9

Pizza: $10

Hot Dog: $7

Pho: $13

Steak: $25

Pasta: $18

\*\* Here is your route from O'Hare to Water\_Way on Normal train. \*\*

[ O'Hare-> Jefferson\_Park-> Forest\_Park-> The\_Loop-> United\_Center-> Chicago-> Belmont-> Wrigley\_Field-> Midway-> Soldier\_Field-> U\_Of\_Chicago-> Water\_Way ]

Travel Summary (Normal Train):

Normal Train runs every 15 minutes.

Has Bathroom?: Yes

Chance of Bathroom being full: 41%

Estimated Cost: $6.75

Chance of Boarding: 90%

Chance of Getting a Seat: 55%

Duration of Travel: 85 minutes

Estimated Time of Arrival: 07:55

Menu for food on Normal Train:

Burger: $9

Pizza: $10

Hot Dog: $7

Here is some information about your end stop.

Name of Stop: Water\_Way

Station Type: Park

Has Bathroom?: Yes

Food at Stop: Fancy Food

However, if the user selects “All” for the type of train, and selects either a start or end stop that a type of train does not stop at, the program will simply not print out any information for that type of train, and instead only print out the information for the type of trains that will visit both stops. For example, if we want to go from O’Hare to Juice WRLD Memorial at 8:30 AM on a Thursday, here is what the “All” selection will print out:

Please select the type of train to check your travel:

(Enter Express, Tour, Normal or All)

All

Start Stop:

O'Hare

End Stop:

Juice\_WRLD\_Memorial

Enter your time of departure in a 24-hour format.

Ex) 1:52PM -> 13:52, 6:30AM -> 06:30

08:30

Enter your day of departure. (One day)

Ex) Monday, Friday, Saturday...

Thursday

{Express train does not stop at start or end stop. No route is created.}

\*\* Here is your route from O'Hare to Juice\_WRLD\_Memorial on Tour train. \*\*

[ O'Hare-> Jefferson\_Park-> United\_Center-> Belmont-> Wrigley\_Field-> Soldier\_Field-> Water\_Way-> White\_Sox-> Juice\_WRLD\_Memorial ]

Travel Summary (Tour Train):

Tour Train runs every 45 minutes.

Has Bathroom?: Yes

Chance of Bathroom being full: 43%

Estimated Cost: $10.0

Chance of Boarding: 90%

Chance of Getting a Seat: 77%

Duration of Travel: 90 minutes

Estimated Time of Arrival: 10:00

Menu for food on Tour Train:

Burger: $9

Pizza: $10

Soup: $5

Turkey: $14

Hot Dog: $7

Pork Chop: $17

\*\* Here is your route from O'Hare to Juice\_WRLD\_Memorial on Normal train. \*\*

[ O'Hare-> Jefferson\_Park-> Forest\_Park-> The\_Loop-> United\_Center-> Chicago-> Belmont-> Wrigley\_Field-> Midway-> Soldier\_Field-> U\_Of\_Chicago-> Water\_Way-> White\_Sox-> South\_Side\_Business-> Juice\_WRLD\_Memorial ]

Travel Summary (Normal Train):

Normal Train runs every 15 minutes.

Has Bathroom?: Yes

Chance of Bathroom being full: 24%

Estimated Cost: $9.0

Chance of Boarding: 90%

Chance of Getting a Seat: 62%

Duration of Travel: 108 minutes

Estimated Time of Arrival: 10:18

Menu for food on Normal Train:

Burger: $9

Pizza: $10

Hot Dog: $7

Here is some information about your end stop.

Name of Stop: Juice\_WRLD\_Memorial

Station Type: Park

Has Bathroom?: No

Food at Stop: No Food

Regardless of what type of route the user receives, the program will ask the user whether they want a different path with different start or end stops or if they want to quit the program at the end:

Would you like to get a different path with different stops or quit?

Enter `Again` to get a different path or `Quit` to quit program.

If the user enters ‘Again’, the program will go back all the way to the step of choosing the type of train and do the whole process again. If the user enters ‘Quit’, the program will quit with a message.

Thank you for using the Metro navigator. We hope to see you again!.

**Correct and Efficient use of inheritance with at least one superclass and two subclasses**

We correctly and efficiently used inheritance with our superclass Train and the subclasses ExpressTrain, NormalTrain, and TourTrain. Here is a prime example of how we correctly used inheritance to get something like food information in a much more efficient way for the trains with food on them. First, we make the method in our superclass Train to have a base getFoodInfo and addFoodInfo method to make it easier to put the information from our subclasses in.

public String getFoodInfo(HashMap<String, Integer> foodInfo) {

String foodResult = "";

for (String i : foodInfo.keySet()) {

foodResult += "" + i + ": " + "$" + foodInfo.get(i) + "\n";

}

return foodResult;

}

public HashMap<String, Integer> addFoodInfo() {

foodInfo.put("Burger", 9);

foodInfo.put("Pizza", 10);

foodInfo.put("Hot Dog", 7);

return foodInfo;

}

Then, in each of our subclasses, we make sure that it extends the Train class:

public class NormalTrain extends Train {

public class ExpressTrain extends Train {

public class TourTrain extends Train{

Since the ExpressTrain does not have any food, it will not use getFoodInfo or addFoodInfo. NormalTrain will use the superclass methods as follows:

public String getTourFoodInfo() {

HashMap<String, Integer> tourTrainFood = super.addFoodInfo();

return super.getFoodInfo(tourTrainFood);

}

TourTrain will use the superclass addFoodInfo to make adding special menu items only for the weekend much easier, while also making changes to the menu much easier.

public String getTourFoodInfo() {

HashMap<String, Integer> tourTrainFood = super.addFoodInfo();

// Different menu by day

if (dayOfDeparture.toLowerCase().equals("saturday") || dayOfDeparture.toLowerCase().equals("sunday")) {

tourTrainFood.put("Lobster", 55);

tourTrainFood.put("Bulgogi", 25);

tourTrainFood.put("Lamb Chop", 35);

} else if (dayOfDeparture.toLowerCase().equals("monday") || dayOfDeparture.toLowerCase().equals("wednesday") || dayOfDeparture.toLowerCase().equals("friday")) {

tourTrainFood.put("Pasta", 18);

tourTrainFood.put("Pho", 13);

tourTrainFood.put("Steak", 25);

} else {

tourTrainFood.put("Soup", 5);

tourTrainFood.put("Turkey", 14);

tourTrainFood.put("Pork Chop", 17);

}

return super.getFoodInfo(tourTrainFood);

}

This brings us to the prompt question:

**Why is inheritance useful for your previously specified superclass and subclasses?**

Inheritance is useful for the superclass Train along with their subclasses TourTrain, ExpressTrain, and NormalTrain because if we have these set methods for each of our trains, such as calculating the cost of a trip, or the percentage chance of getting on a train, it takes away a substantial amount of duplicate code that we otherwise would have had without this superclass. This made it much more organized for us to work with, especially as our code got bigger, with all sorts of different methods. Furthermore, since there were methods that derived from our superclass, it made it much more flexible for us to make quick changes, and not have to worry about each subclass. For example, during our coding, we decided that we wanted to change the way that we calculated estimated arrival times for a train. Without a superclass, we would have had to gone to each train type and change the calculation of estimated arrival times one by one; however, with a superclass, all we had to do was change a couple of lines of code, and that superclass method was able to apply itself across all our subclasses in an efficient way. Here is what our final method for calculating estimated arrival time looked like in our superclass:

public String calculateArrivalTime(List<Stop> stopsList, int timeTaken) {

LocalTime now = LocalTime.parse(timeOfDeparture);

return now.plusMinutes(timeTaken).toString();

}

This leads to a great increase in code reusability, which can also be seen in overriding methods.

The usage of inheritance also allowed us to use overriding for the methods in subclasses. Overriding allows us to classify a behavior that is specific to the subclass, and the subclass can implement a superclass method based on its necessity. One of the examples we effectively implemented overriding was the method to calculate the cost. The `calculateCost` method in the superclass `Train` only calculates the cost by one case, which is to just add the cost by the number of stops travelled. However, the logic needed to be done differently in subclasses `ExpressTrain` and `TourTrain` because each train had a different way of calculating the cost. Below is an example of the overriding `calculateCost` method in the subclass `ExpressTrain`.

@Override

public Double calculateCost(List<Stop> stopsList) {

int numExpressStops = 0;

for (Stop stops : stopsList) {

String stopType = stops.findData("TypeOfStop");

if (stopType.equals("Express") || stopType.equals("All")) {

numExpressStops += 1;

}

}

// Start stop does not count

numExpressStops -= 1;

// Add Base Price by number of stops

if (numExpressStops < 3) {

return 5.0;

} else {

return 8.0;

}

}

Instead of increasing the cost per stop, the logic checks the number of stops and decides the cost by the number. Similarly, `calculateCost` works differently in the `TourTrain` subclass and differs from the logic used in that of superclass `Train`. `NormalTrain` subclass does not override, but inherits the `calculateCost` method from the superclass `Train` because it follows the same logic. Therefore, overriding made the methods more specific for subclasses when necessary.

Using the same name and overriding makes the code more readable because the goal of the method is already present in the superclass while the logic specific to each subclass can be distinguished. The frequent usage of overriding and inheriting superclass methods for the usage in subclass methods allows other people who are not well aware of the code to look at the methods in subclasses and immediately know what the methods try to achieve. Naturally, inheritance also led to making our code more readable.

**Correct and Efficient use of two other class concepts**

We correctly and efficiently used the graph code from HW8 to establish the stops as vertices and the links between stops as edges, along with the usage of queue to implement breadth-first search to eventually find the shortest path possible from one stop to another. This brings us to the following prompt question: **For each of the two additional previously-specified class concepts that you used, why is that concept the best to use in your project?**

Our project is very similar to the HW8 in that we have nodes (stops) and they need to be directly connected to each other so that we can eventually get the shortest path from one stop to another. Subsequently, we felt that using a graph to represent the nodes as vertices and the links between them as edges was the best strategy, so that we could use breadth-first search, which is the most efficient way that we learned to find a shortest path between nodes. Queue was used to implement breadth-first search because it was the most efficient for breadth-first search. Queue is a first in first out structure, which allows us to start with the root node and check the neighboring nodes in order. The neighboring nodes will be enqueued so that their neighboring nodes can also be visited. This keeps track of nodes remaining to be visited to find the shortest path. As long as the queue is not empty, we will get to visit all the nodes in the graph.

Therefore, graph and queue were the best class concepts to use in our project, to organize train stops and their links and vertices and edges, and implement breadth-first search to find the shortest path from one stop to another.

**Sufficiently substantive project**

**What does your project do that is interesting and substantive?**

At first, we were admittedly worried that our project may be too similar to HW8; however, we decided to add many different aspects that a user would find useful if they were to actually go traveling using our program. The first thing that we knew we needed to do was add multiple options for the user to potentially choose from, as that would add more complexity to the project, and we also felt that it was applicable in the real world. When traveling, people have different priorities for what brings them the most comfort. Some people may be struggling financially, so the reduced ticket cost is enough to sacrifice 20 minutes of their day to get to a stop, or maybe a person is willing to pay the extra money to get to their destination faster. Either way, by giving the choice to the user to view all trains, it allows for them to choose which scenario is best for them. Furthermore, to even add more real-life applicability, we decided to give data on different things that people may care about; such as the chances that a seat will be available, the chances that the bathroom will be available, or most importantly, the chances that a user can actually get on a train. That brings us to the next complexity we added to make the project substantive and interesting, which is that depending on the day and time of the departure, different aspects change, such as how popular each train will be, which affects the probability of someone getting on the next train. We considered the fact that it is harder to get on a train during rush hour. Also, the estimated time of arrival in the console only shows only when the time of departure is within the operating time of the train. Such consideration makes the project more real and interesting. Finally, we also added the aspect of different types of food on each train, which we felt was appropriate for each train type. The type of food also changes based on the day, which adds real-life applicability, as that is essentially special menu items being added for certain times, which may raise the incentive for people to take that particular type of train.

**README complete and clear**

The README is completed in a correct and well-organized manner, including an overview section that describes what the program does and small examples of the code in necessary places to explain its functionality.

**Good style and organization, including JavaDocs style methods and comments**

We have good style and organization throughout our code, including Javadocs style methods and comments. The code is in a sufficient JavaDocs-style because there are comments that describe the class, its methods and parts of code that may need a brief explanation. Here is an example of one code chunk to show our organization:

/\*\*

\* Check if departure time lies in the range of train operating time

\*

\* @param time string that represents time

\* @return boolean that represents whether the train runs at departure time

\*/

public boolean checkTrainTime(String time) {

LocalTime now = LocalTime.parse(time);

// Start time for train

LocalTime morning = LocalTime.parse("05:30");

// End time for train

LocalTime night = LocalTime.parse("23:30");

if (now.isBefore(morning) || now.isAfter(night)) {

return false;

}

return true;

}

It also follows the style guide posted on moodle for the usage of capitalization for names of class and variable, file organization for the order of import statements, Javadoc comments, data fields, constructors and main. It also includes necessary comments in a correct format, uses descriptive variable and method names (multi-word if necessary) and uses braces and whitespace in an appropriate format. Line length and indentation were also considered for better readability.