Homeaway From Home!

Data Structures and Algorithms
Project, version 1.2 – 2025-10-7
Added text is presented in green
Deleted text appears ~~crossed out~~.

Important remarks

Team this project is to be MADE BY GROUPS OF 2 STUDENTS.

1st Class Diagram Report - 1% Submission of the document to **Moodle** until 3-10-2025. Only use the interfaces and classes in dataStructures given for this first phase.

1st program - 10% Submission and acceptance of the source code to **Mooshak** (contest AED_2025_TP problem A) until 31-10-2025. Only use the interfaces and classes in dataStructures given for this first phase. You must complete the implementation of all classes in this dataStructures package, even if you do not use it. It is not allowed to use the java.util package, except the classes used for writing or reading (e.g. Scanner).

Final Report of the 1st program - 2% Submission of the document to **Moodle** until 31-10-2025.

2nd Class Diagram Report - 1% Submission of the document to **Moodle** until 7-11-2025. Use the interface and classes in dataStructures given for this second phase.

2nd program - 16% Submission and acceptance of the source code to **Mooshak** (contest AED_2025_TP problem B) until 5-12-2025. Only use the interfaces and classes in dataStructures given for this second phase. You must complete the implementation of all classes in this dataStructures package, even if you do not use it. It is not allowed to use the java.util package, except the classes used for writing or reading (e.g. Scanner). The text processing related to the command "tag" must be implemented by you, without using the methods of the class String.

Final Report of 2nd program - 5% Submission of the document to **Moodle** until 5-12-2025.

A submission will only be evaluated if it follows the rules mentioned above. Programs submitted to Mooshak will only be evaluated if there is at least one submission of that program to Mooshak with 100 points (pass all tests). For programs submitted to Mooshak with 100 points, the functionality has a weight of 20%, and the quality of the code has a weight of 80% in the grade of the program. The 20% of functionality is only guaranteed if the program follows the mentioned rules (there may be penalties, for example, using the java.util package). At each stage of the work (1st and 2nd programs) there will be an oral discussion, which will have a grade between 0 and 20 values. The final grade for a stage is the minimum between the grades of the submitted work and oral discussion.

1 Development of the application *HomeAway From Home*

1.1 Problem description

The goal of this project is to develop an application that helps international students in a campus-oriented town by providing information about services that are useful to them. These services are location-based (which means that they may be referred to as locations) and may be of various kinds, which will be defined by the particular application. We will provide you with examples for Lisbon's Metropolitan area, which includes Almada, Monte da Caparica and Costa de Caparica, where several universities and Higher Education Schools exist, but the application should be useful for any such a town.

The application works, at a given time, with a single area but can have access to several areas that are saved in secondary memory (files with extension ".ser"). The areas saved in the file are in ~~the "data" folder~~ the root folder, which is ~~in~~ the executable program directory.

For the sake of this assignment, the type of services will be *Eating* (such as low-budget restaurants or canteens) and *Lodging* (rooms that can be used by students and in student residences) as well as *Leisure* (Entertainment/Cultural) because student life is not just about saving money. Each area has on average around 2500 services, and in this application, services are never deleted from the area. New services are added regularly, as these student areas are evolving rapidly.

The application should be geographically oriented, which means that the town is defined by a bounding box (rectangle) and should only accept new services included in that bounding box.

Each service will always include a name and a geographic location, which is composed of latitude and longitude (for simplicity, in our system, latitude and longitude are integer (long) numbers that represent the real decimal values multiplied by 1 000 000 000). Each type of service will include additional information. An *Eating* service will include the cost of the daily student menu and the number of seats. The *Lodging* type of service will include the monthly cost of the room, for students, and the number of single rooms. The *Leisure* type will include the ticket price and the percentage of student discount. **Be aware that the system's input and output does only handle integer values (although you may want to use float values for internal computations).**

The system should also enable diverse types of students. For simplicity, we will consider *Bookish* students (mainly concerned with studying and visiting/attending *Leisure* sites and/or events), *Outgoing* students (who are mainly concerned with eating out and visiting the town) and *Thrifty* students (whose main concern is to save money and live on a budget). Students should only go or move to locations inside the bounding box. At a particular moment, the location of a student may be their current home (a lodging service) or any services known by the system. A bookish student collects information about the leisure places that they have visited in the town. Outgoing students collect information about all the locations/services they have visited, with no restrictions. Thrifty students are always trying to find the least expensive place to eat and sleep. They always seek for the least expensive canteen/restaurant and residence/hostel in town and will only move from their current home if they find a cheaper service than the one they currently use.

The number of students in each area is around 1200, however they can enter and leave the area whenever they wish. Everything that involves the movement of students, entry, exit and visits to places is very frequent, as it involves a very active community.

Students may evaluate services that they have visited, stayed in or ate at. Evaluations are submitted as stars and each service can provide its updated evaluation, which is the integer rounded result of the average of all the evaluations received. A freshly created service is given 4 stars. Listings of locations ordered by star evaluation may be requested by the system. A

particular student may also require to know the most relevant service (for them) of particular type.

The search for a service of a given type with a given evaluation that is as close as possible to the location of a given student is one of the most used features in the application.

The system will know all the services inside the defined bounding box, all registered students, all students in a given lodging or eating service, and the evaluations of a given service.

2 Commands

In this section we present all the commands that the system must be able to interpret and execute. In the following examples, we differentiate `text written by the user` from the feedback written by the program in the console. You may assume that the user will make no mistakes when using the program other than those described in this document. In other words, you only need to take care of the error situations described here, in the exact same order as they are described.

For each command, the program will only produce one output. The error conditions of each command have to be checked in the exact same order as described in this document. If one of those conditions occurs, you do not need to check for the other ones, as you only present the feedback message corresponding to the first failing condition. However, the program does need to consume all the remaining input parameters, even if they are to be discarded.

Commands are case insensitive. For example, the `exit` command may be written using any combination of upper and lowercase characters, such as `EXIT`, `exit`, `Exit`, `exIT`, and so on. In the examples provided in this document, the symbol ↵ denotes a change of line.

If the user introduces an unknown command, the program must write in the console the message Unknown command. Type help to see available commands. For example, the non existing command `someRandomCommand` would have the following effect:

```
someRandomCommand↵
Unknown command. Type help to see available commands.↵
```

If there are additional tokens in the line (e.g. parameter for the command you were trying to write, the program will try to consume them as commands, as well. So, in the following example, `someRandom Command` would be interpreted as two unknown commands: `someRandom` and `Command`, leading to two error messages.

```
someRandom Command↵
Unknown command. Type help to see available commands.↵
Unknown command. Type help to see available commands.↵
```

Several commands have arguments. Unless explicitly stated in this document, you can assume that the user will write the correct number of arguments, and with the correct type. However, some of those arguments may have an incorrect value. For that reason, we need to test each argument exactly by the order specified in this document. Arguments will be denoted **`with this style`**, in their description, for easier identification. Also, for any String arguments in the commands, assume they are case-insensitive. So, "Coffee Shop" and "coffee shop" would be the same service, for example. However, **you must be careful**, since when writing this information, it should always be written as it was created (service name in the bounds command, student name in the student command). Note that the service and student types will always be written in lowercase.

2.1 `exit` command

Terminates the execution of the program. This command does not require any arguments. The following scenario illustrates its usage.

```
exit↵
Bye!↵
```

This command always succeeds. If there is a current geographic area, it should be saved in a text file, whose name is the same as the name of the area, where the space character is replaced by "_".

2.2 `help` command

Shows the available commands. This command does not require any arguments. The following scenario illustrates its usage.

```
help↵
bounds - Defines the new geographic bounding rectangle↵
save - Saves the current geographic bounding rectangle to a text file↵
load - Load a geographic bounding rectangle from a text file↵
service - Adds a new service to the current geographic bounding rectangle. The service may be eating, lodging or leisure↵
services - Displays the list of services in current geographic bounding rectangle, in order of registration↵
student - Adds a student to the current geographic bounding rectangle↵
students - Lists all the students or those of a given country in the current geographic bounding rectangle, in alphabetical order of the student's name↵
leave - Removes a student from the the current geographic bounding rectangle↵
go - Changes the location of a student to a leisure service, or eating service↵
move - Changes the home of a student↵
users - List all students who are in a given service (eating or lodging)↵
star - Evaluates a service↵
where - Locates a student↵
visited - Lists locations visited by one student↵
ranking - Lists services ordered by star↵
ranked - Lists the service(s) of the indicated type with the given score that are closer to the student location↵
tag - Lists all services that have at least one review whose description contains the specified word. This list is from the most recent review to the oldest↵
find - Finds the most relevant service of a certain type, for a specific student↵
help - Shows the available commands↵
exit - Terminates the execution of the program↵
```

This command always succeeds. When executed, it shows the available commands.

2.3 `bounds` command

Defines the geographic bounding rectangle of the system. The command receives, as arguments, the **`name`** of the area to be considered in the system and its bounding rectangle,

composed of the **latitude** and the **longitude** of the top left and bottom right corners of this bounding rectangle (in this order). The values will be integers. This command fails if:

- An area with the same name already exists (the current one or saved in a file). In this case, the system will return the message (Bounds already exists. Please load it!).
- If the defined bounds rectangle is not valid - if the top latitude is lower than or equal to the bottom latitude and the right longitude is lower than or equal to the left longitude - it is not possible to include any services in the defined rectangle. In this case, the system will return the message (Invalid bounds.).

Whether the command succeeds, if there is a current area, it must be saved to a text file (whose name is the same as the name of the area, where the space character is replaced by "_"), the new area becomes the current area, and the feedback message is (<bounds name> created.).

In the example, we define the bounding rectangle for the called **Lisbon Area**.

```
bounds 38848911991 -9497899738 38461247511 -8885274394 Lisbon Area↵
Lisbon Area created.↵
```

Note that if the command fails, the current area remains. If there is no current area, the system cannot really do anything, so execution of any command will generate the message (System bounds not defined.).

2.4 `save` command

Saves the current geographic bounding rectangle to a text file. The command does not require any arguments. If there is a current area, it must be saved to a ~~text~~ file, the save area remains the current area, and the feedback message is (<bounds name> saved.). The name of the file is the same as the name of the area, where the space character is replaced by "_"). In the example, we save the current bounding rectangle.

```
bounds 38848911991 -9497899738 38461247511 -8885274394 Lisbon Area↵
Lisbon Area created.↵
...
save↵
Lisbon Area saved.↵
```

This command fails if there is no a current area. In this case, the system will return the message (System bounds not defined.).

2.5 `load` command

Load a geographic bounding rectangle of the system. The command receives, as arguments, the **name** of the area to be considered in the system. This command fails if:

- There is no ~~text~~ file in the ~~"data"~~ root folder with the same name as the given area name where the space character was replaced by "_". In this case, the system will return the message (Bounds <area name> does not exists.).

Whether the command suceeds, if there is a current area, it must be saved to a ~~text~~ file, the load area becomes the current area, and the feedback message is (<bounds name> loaded.).

In the example, we load the bounding rectangle called **Lisbon Area**.

```
load Lisbon Area↵
Lisbon Area loaded.↵
```

Note that if the command fails, the current area remains. If there is no current area, the system cannot really do anything, so execution of any command will generate the message (System bounds not defined.).

2.6 `service` command

Adds a new service of a given type to the current geografic area. The command receives, as arguments, the **type** of service, the **latitude** and the **longitude** of the service, the **price** of the service, the **value** of the service and the **name** of the service. The **type** of service may be: eating, lodging or leisure. The **price** may be the price of the menu for the eating service, the price of the room for the lodging service, or the price of the ticket for the leisure service. The **value** may be the capacity for eating and lodging services, and the student service discount for leisure service. When successful, the program will output the feedback message (<service type> <service name> added.).

In the example, we create a new eating service called **Akatsuki Caparica** at (38.639378483, -9.234400975) with the price of the student menu at **8** euros, and the maximum capacity equal to **50**. There is no upper limit to the number of services (of any type) one can create in the system.

```
service eating 38639378483 -9234400975 8 50 Akatsuki Caparica↵
eating Akatsuki Caparica added.↵
```

In the example, we create a new lodging service called **Frausto da Silva Residency** at (38.665184410, -9.208892510) with the monthly price of the student room at **251** euros, and the maximum capacity equal to **80** . There is no upper limit to the number of services (of any type) one can create in the system.

```
service lodging 38639378483 -9234400975 251 80 Frausto da Silva
Residency↵
lodging Frausto da Silva Residency added.↵
```

In the example, we create a new leisure service called **Joaquim Benite Theater** at (38.676672199, -9.159911717) with the ticket price of **12** euros and the student discount of **20**%. There is no upper limit to the number of services (of any type) one can create in the system.

```
service leisure 38676672199 -9159911717 12 20 Joaquim Benite Theater↵
leisure Joaquim Benite Theater added.↵
```

The following errors may occur:

1. If the **type** is not a valid service type, the error message is (Invalid service type!).
2. If the service location is outside the bounding rectangle defined for the system, the error message is (Invalid location!).

3. If the **price** is less or equal to **0**, the error message is (Invalid menu price!) for eating service; (Invalid room price!) for lodging service; (Invalid ticket price!) for leisure service.

4. If it is a leisure service and the **discount** is less than **0** or greater than **100**, the error message is (Invalid discount price!).

5. If it is a eating or lodging service and the **capacity** is less or equal to **0**, the error message is (Invalid capacity!).

6. If the service **name** already exists in the current geography area, the error message is (<service name> already exists!).

2.7 `services` command

Displays the list of services in the system. The command does not receive any arguments and always succeeds. If there are no services to list, the program will output the feedback message (No services yet!). Otherwise, it lists all services, in order of insertion, each line presenting the service name, its type, latitude and longitude (<service name>: <type> (<latitude>, <longitude>).). In the example, there are three services available in the system.

```
services↵
Akatsuki Caparica: eating (38639378483, -9234400975).↵
Frausto da Silva Residency: lodging (38639378483, -9234400975).↵
Joaquim Benite Theater: leisure (38676672199, -9159911717).↵
```

2.8 `student` command

Adds a student to the system. The command receives as arguments the **type** of student (bookish, outgoing or thrifty), their **name**, their origin **country**, and the name of the **lodging** where they are living currently. When successful, the program will output the feedback message (<student name> added.). We assume that the student's location, at creation, is their lodging service. In the example, we create a new bookish student called **Neil Perry**.

```
student bookish↵
Neil Perry↵
United States of America↵
Frausto da Silva Residency↵
Neil Perry added.↵
```

The following errors may occur:

1. If the student's **type** is not valid the error message is (Invalid student type!).

2. If the **lodging** does not exist in the system, the error message is (lodging <lodging name> does not exist!).

3. If the lodging service is already full, the error message is (lodging <location name> is full!).

4. If the student's **name** already exists in the system, the error message is (<student name> already exists!).

2.9 `leave` command

Removes a student from the system. The command receives as argument the student's **`name`**. When successful, the program will output the feedback message (<student name> has left.). In the example, we remove student **`Neil Perry`**.

```
leave Neil Perry↵
Neil Perry has left.↵
```

The following errors may occur:

1. If the student's **`name`** does not exist in the system, the error message is (<name> does not exist!).

2.10 `students` command

Lists all the students or those of a given country. The command has one argument (the word "all" or the **`name`** of a country), and always succeeds. If there are no students to list, the program will output the feedback message (No students yet!) in case of "all", or (No students from <name>!). Otherwise, it lists all the students if the argument is "all" or the students of the given country. The student information appears, in alphabetical order in case of all students, or in registation order in the case of a given country. Each line presents the students's name, their type and their current location's name.

In the following example, there are three students in the system.

```
students all ↵
Neil Perry: bookish at Frausto da Silva Residency.↵
Knox Overstreet: outgoing at Joaquim Benite Theater.↵
Todd Anderson: thrifty at NOVA FCT Canteen.↵
```

In the following example, there is one student in the system from Australia.

```
students Australian ↵
Knox Overstreet: outgoing at Joaquim Benite Theater.↵
```

2.11 `go` command

Changes the location of a student to a eating service, or leisure service. The command receives as arguments the student's **`name`** and the name of the **`location`** where the student is going. This location can either be a eating service or a leisure service in the system. Bookish students store the leisure services they have visited and outgoing students store every service they have visited. Thrifty students do not store visited locations. When successful, the program will output the feedback message (<student name> is now at <location name>.). If the student is thrifty, and the command is moving their location to a eating service more expensive than the cheapest they have visited so far, the command is accepted but the student is warned of their mishap (<student name> is distracted!).

We provide two example: **`Todd Anderson`** goes to eat at **`Akatsuki Caparica`**, which is not their less expensive known eating location; and **`Todd Anderson`** goes to a leisure service **`Caparica event`**.

```
go Todd Anderson↵
Akatsuki Caparica↵
Todd Anderson is now at Akatsuki Caparica. Todd Anderson is distracted!↵
go Todd Anderson↵
Caparica Event↵
Todd Anderson is now at Caparica Event.↵
```

The following errors may occur:

1. If the name of the **location** where the student is going is not known to the system, the error message is (Unknown <location name>!).
2. If the student's **name** does not exist in the system, the error message is (<student name> does not exist!).
3. If the **location** where the student is going is not an eating or a leisure service, the error message is (<location name> is not a valid service!).
4. If the student decides to go to the location where they are already, the command has no effect but returns the error message (Already there!).
5. If is an eating service and it is already full, the error message is (eating <location name> is full!).

2.12 move command

Changes the home of a student if that is acceptable. The command receives as arguments the student's **name** and the name of the **lodging** service to become the student's new home. When successful, the student's home becomes the lodging service submitted and the student's current location also becomes the location of this service. The program will output the feedback message(lodging <lodging name> is now <student name>'s home. <student name> is at home.). In the example, **Neil Perry** moves to **Egas Moniz University Residence**.

```
move Neil Perry↵
Egas Moniz University Residence↵
lodging Egas Moniz University Residence is now Neil Perry's home. Neil Perry is at home.↵
```

The following errors may occur:

1. If the **lodging** service name does not exist in the system, the error message is (lodging <lodging name> does not exist!).
2. If the student's **name** does not exist in the system, the error message is (<student name> does not exist!).
3. If the student's home is already the one provided in the command, the system should do nothing and only provide the message (That is <student name>'s home!).
4. If the lodging service is already full, the error message is (lodging <location name> is full!).
5. If a thrifty student tries to move to a same price or more expensive lodging, the error message is (Move is not acceptable for <student name>!).

2.13 `users` command

Lists all students currently in a given eating or lodging service. The command receives as arguments the service's **`name`**, the **`order`** desired (may be ">" - from oldest to newest based on insertion order; or "<" - from newest to oldest based on insertion order into service), and the name of the eating or lodging service. If there are no students to list, the program will output the feedback message (No students on <service name>!). Otherwise, it lists all the students at the given service, in the desired order. Each line presenting the students's name and their type.

In the following two examples, there are three students at the lodging **Egas Moniz University Residence**.

```
users > Egas Moniz University Residence ↵
Neil Perry: bookish↵
Knox Overstreet: outgoing↵
Todd Anderson: thrifty↵
users < Egas Moniz University Residence ↵
Todd Anderson: thrifty↵
Knox Overstreet: outgoing↵
Neil Perry: bookish↵
```

The following errors may occur:

1. If the order type is neither ">" nor "<", the error message is (This order does not exists!).
2. If the service name does not exist in the system, the error message is (<service name> does not exist!).
3. If the given service is neither eating nor lodging service, the system provide the message (<service name> does not control student entry and exit!).

2.14 `where` command

Locates a student. The command receives as argument the student's **`name`**. When successful, the program will output the feedback message (<student name> is at <location name> <location type> (<location latitude>, <location longitude>).). In the example, student **`Todd Anderson`** is at **`Cinemas NOS Almada Forum leisure (38661501872, -9174040540)`**.

```
where Todd Anderson↵
Todd Anderson is at Cinemas NOS Almada Forum leisure (38661501872, -9174040540).↵
```

The following error may occur:

1. If the student's **`name`** does not exist in the system, the error message is (<student name> does not exist!).

2.15 `visited` command

Lists the locations visited and stored by one student. The command receives as argument the student's **`name`** and, if it is the name of a bookish or outgoing student, it lists the locations stored by them. Thrifty students do not store the locations they visited. When successful, the program will list the names of the locations visited by the student, one in each

line, sorted by visiting order. In the example, leisure places visited by bookish student **Neil Perry** are listed.

```
visited Neil Perry↵
Cinemas NOS Almada Forum↵
Joaquim Benite Theater↵
```

The following errors may occur:

1. If the student's **name** does not exist in the system, the error message is (<student name> does not exist!).
2. If the student is thrifty, the message provided will always be (<student name> is thrifty!).
3. If the student has not visited any locations, the message provided will be (<student name> has not visited any locations!).

2.16 `star` command

Evaluates a service. The command receives as arguments an integer **n** between 1 and 5, representing the number of stars attributed to the service, followed by the service's **name** and evaluation **description**. Evaluations are anonymous. The system should maintain the average of the evaluation provided, including the initial one, attributed at the time of creation. When successful, the program will output the feedback message (Your evaluation has been registered!). In the example, **Egas Moniz University Residence** received a vote of **4** with a description **Good residence!**.

```
star 4 Egas Moniz University Residence↵
Good Residence!↵
Your evaluation has been registered!↵
```

The following errors may occur:

1. If the number **n** is not between 1-5, the error message is (Invalid evaluation!).
2. If the service's **name** does not exist in the system, the error message is (<name> does not exist!).

2.17 `ranking` command

Lists all services ordered by star. The command receives no arguments. It prints a header (Services sorted in descending order) and then lists all services in the system, sorted in descending order, by evaluation average. Each printed line should include the message (<service name>: <integer rounded evaluation average>.). For services with the same average, the order of printing should be the order by which the average was last updated by the service. If there are no services in the system, the header should be replaced by the message (No services in the system.).

```
ranking↵
Services sorted in descending order↵
Cinemas NOS Almada Forum: 4↵
Joaquim Benite Theater: 4↵
Akatsuki Caparica: 3↵
NOVA FCT Canteen: 2↵
```

ATTENTION: The evaluation average value is an integer resulting from rounding using Math.round(float value).

2.18 `ranked` command

Lists the service(s) of the indicated type with the given score that are closer to the student location. The command receives as arguments a **n** star average, the **type** of service and the student's **name**. When successful, the program will print a header line (<type> services closer with <n> average) followed by the name of the closest service. In the event of a tie in distance, a list of service names is displayed, one per line, in order of the time at which the current average star rating was obtained.

The system uses the Manhattan Distance for computing the distance between two locations. Namely, given two locations l^1 and l^2, where (l^1_{lat}, l^1_{long}) and (l^2_{lat}, l^2_{long}) denote their corresponding latitude l^*_{lat} and longitude l^*_{long} coordinates, the distance should be:

$$d(l^1, l^2) = |l^1_{lat} - l^2_{lat}| + |l^1_{long} - l^2_{long}| \tag{1}$$

where $|\cdot|$ denotes the absolute value.

In the example, **leisure** services with a **4** average, which is closer to the student **Neil Perry** location are printed.

```
ranked leisure 4 Neil Perry↵
leisure services closer with 4 average↵
Cinemas NOS Almada Forum↵
Joaquim Benite Theater↵
```

The following errors may occur:

1. If **n** is larger than **5** or lower than **1**, the error message is (Invalid stars!).
2. If the student's **name** does not exist in the system, the error message is (<student name> does not exist!).
3. If the service **type** is not "eating", "lodging" or "leisure", the error message is (Invalid service type!).
4. If no services of the **type** exist, the error message is (No <type> services!).
5. If no services of the **type** exist with **n** star average, the error message is (No <type> services with average!).

2.19 `tag` command

List all services that contains the specified tag in reviews. The command receives as arguments the **`tag`**. When successful, the program will list the name of all services that have at least one evaluation with this tag (case-insensitive, that is, "Good" is the same as "good" or "GOOd"), in order of insertion of services. Each line in the output consists simply of the name of the service and its type.

If there is no service with a review that contains the tag, the message provided will be (There are no services with this tag!).

In the example, services that have at least one review with **`tag`** are printed.

```
tag good↵
leisure Cinemas NOS Almada Forum↵
lodging Joaquim Benite Hostel↵
```

2.20 `find` command

Finds the most relevant service of a certain type, for a specific student. The command receives as arguments the student's **`name`** and the **`type`** of service required. Depending on the type of the student, the system may provide the best service (with best average) of the type (for bookish and outgoing students), or the less expensive one (for thrifty students). In the case of ties (several services with the same average points or price), the system should provide the service with more time in this average or the first service inserted in the system, respectively.

When the find command is successful, the program displays a message with the (<service name>) of the service provided by the system.

We provide two successful examples. In the first example, **`Neil Perry`**, a bookish student, searches for the best **`eating`** service, and receives NOVA FCT Canteen. In the second example, **`Charlie Dalton`**, a thrifty student, searches for the least expensive **`lodging`** service, and receives Frausto da Silva Residency.

```
find Neil Perry↵
eating↵
NOVA FCT Canteen↵
find Charlie Dalton↵
lodging↵
Frausto da Silva Residency↵
```

The following errors may occur:

1. If the service **`type`** is not "eating", "lodging" or "leisure", the error message is (Invalid service type!).
2. If the student's **`name`** does not exist in the system, the error message is (<student name> does not exist!).
3. If no services of the specified **`type`** exist in the system, the error message is (No <type> services!).

3 Submission to Mooshak

To submit your project to Mooshak, please register your group in the Mooshak contest AED_2025_TP. The Mooshak login for the group must be the concatenation of the numbers of the students that make up the group, separated by _ (the first number must be the smallest). For example, students #5678 and #5677 should have their Mooshak username **5677_5678**. **Only the projects submitted, for evaluation, through logins following the above rule will be accepted.** The name and number of students that make up the group must be inserted in the header of all submitted files, as follows:

```
/**
* @author STUDENT1NAME (STUDENT1NUMBER) STUDENT1temail
* @author STUDENT2NAME (STUDENT2NUMBER) STUDENT2temail
*/
```