Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies

Object Oriented Programming

Semestral project – EspaTrip desktop application

Petra Miková

Petra Miková
ID: 120852

Contents

Petra Miková
ID: 120852

Project objective

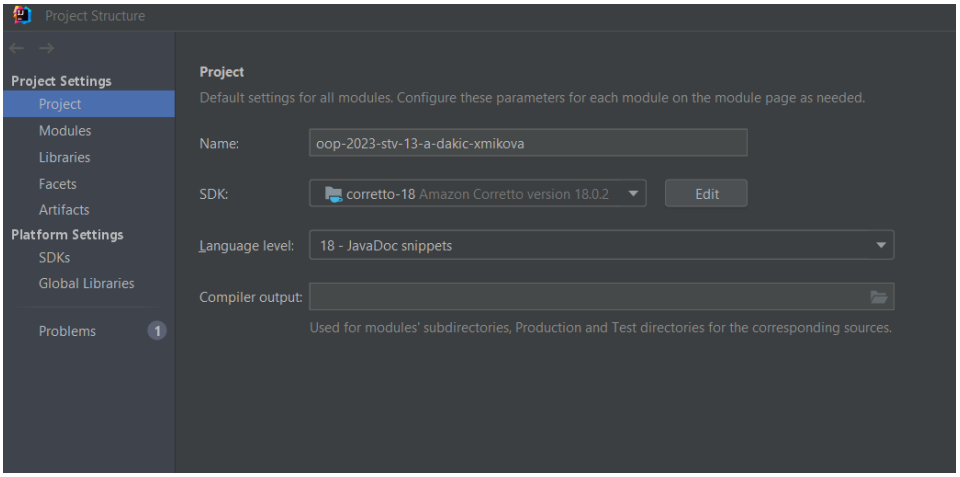
The desktop app EspaTrip allows a user to register as a customer who wants to go on a trip to Spain, or a Spanish local who offers a stay for people from all around the world. For now, there are three favorite Spanish destinations available - Madrid, Barcelona and Valencia.

As a customer - you decide on a city and your preferences on a local, and after that you will be offered to choose from multiple restaurants and sights, depending on what you want to see and what food you want to try. After all this, you get a price offer and a summary of your trip , which you can reject or accept and send in the order. Then your order is pending and if a sufficient local accepts it, you are good to go!

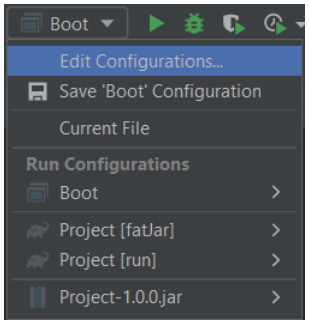
As a local - within your initial sign up, you fill in the registration form created for locals and your information will be stored, and then in your profile you will see all the created trips from customer whose criteria you meet. You can either reject or accept the trip, and if you accept it it will move into your accepted trips part of the profile and you can start preparing for the adventure!

Project requirements

The project runs on Gradle. The Java source compatibility for this project is 18, so it is important that in your IDE you are able to run SDK either Java 18.0.2 or Java Amazon Corretto 18.0.2. Gradle should show you the option to download this SDK if your IDE is not recognizing it or you do not have it set up in the IDE.



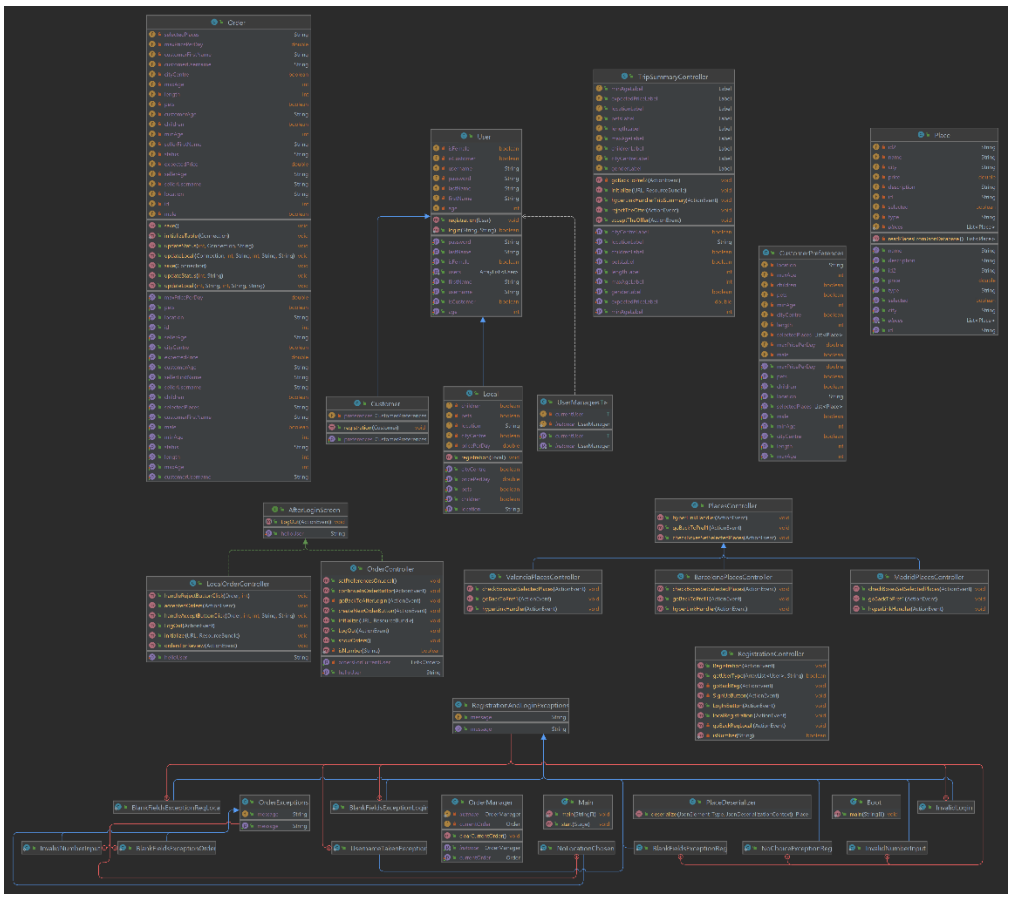
When the Gradle build is all set up, you can either run it with gradle run configuration or directly by running the Boot class which is located in the org.example.app package. To run the JAR file, please build it with fatJar configuration and then it is possible to run it even outside the IDE.



Petra Miková
ID: 120852

UML diagram and classes relationship

UML diagram



Classes relationship

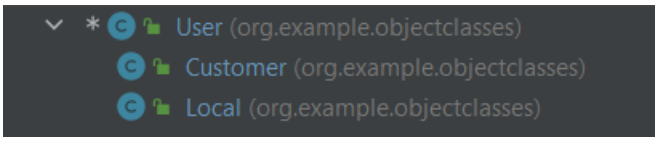
1. The Customer and Local classes inherit from User class
2. The BarcelonaPlacesController, MadridPlacesController, and ValenciaPlacesController inherit from PlacesController
3. The UserManager class uses generic type of subclasses of User class
4. Both the RegistrationAndLoginExceptions and OrderExceptions classes have multiple nested classes that inherit from these super classes and handle specific exceptions. The super classes themselves inherit from Java builtin class Exception.
5. The OrderController and LocalOrderController classes implement the AfterLoginScreen interface.

Criteria fullfilment

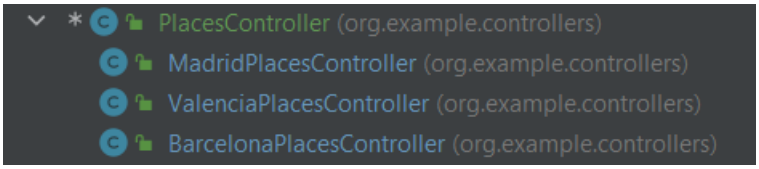
Main criteria

1. Inheritance

First case of inheritance is by classes Customer and Local from super class User, as they use the same functionalities, but have some additional attributes each.



Second case of inheritance is by classes BarcelonaPlacesController, MadridPlacesController, and ValenciaPlacesController, which inherit from PlacesController. They all use the same methods, but have different attributes which represent places in each location.

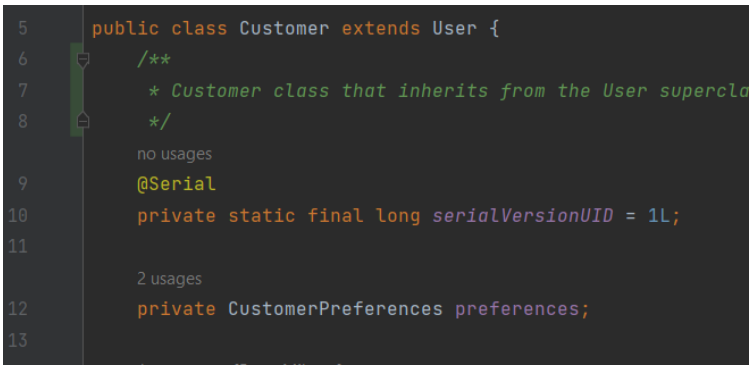


2. Polymorphism

Polymorphism is used in both exceptions classes – RegistrationAndLoginExceptions and OrderExceptions. The default message is „Error“ and then every nested class of those which represents a specific excpetion overrides it and displays its own message based on the specific error.

3. Aggregation

Aggregation is used in the Customer class, where we have an attribute preferences, whose type is CustomerPreferences (another class).



```java
public class CustomerPreferences implements Serializable {
    /**
     * A class to handle all preferences from the customer for his trip.
     */
    2 usages
    private String location;
    2 usages
    private boolean children;
    2 usages
    private boolean pets;
    2 usages
    private boolean male;
    2 usages
    private int length;
    2 usages
    private boolean cityCentre;
    2 usages
    private double maxPricePerDay;
    2 usages
    private int minAge;
    2 usages
    private int maxAge;
    2 usages
    private List<Place> selectedPlaces;
    no usages
    private static final long serialVersionUID = 1L;

```

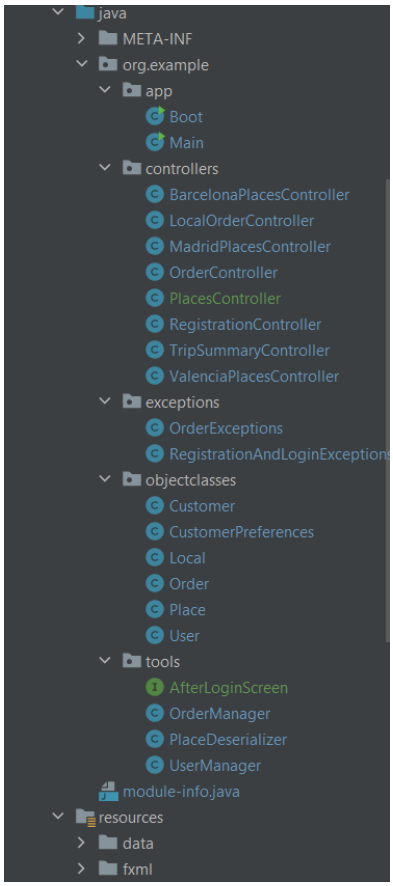
4. Encapsulation

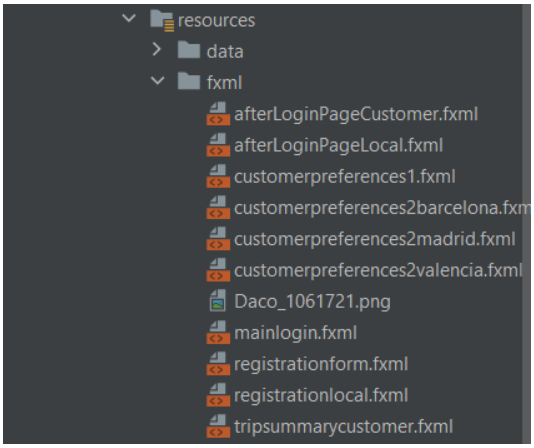
All classes in the org.example.objectclasses use encapsulation – so have private attributes wrapped into public getters and setters. Besides these, the classes UserManager and OrderManager (Singleton classes) use encapsulation as well (org.example.tools).

5. Code organization (packages)

The main package (also a main module) is the org.example. This package is then efficiently organized into 5 packages. The GUI is in resources directory in FXML folder. The app package includes the booting classes. The controllers package includes all the controllers that handle user interaction with GUI. The exceptions package includes all the custom exceptions classes. The package objectclasses includes all the objects used in app – User, Order, Place. Lastly, the tools include other essential miscellaneous classes for handling the functionality like managers using Singleton or class for deserializing JSON database. The FXML in resources includes all the .fxml files for functional GUI of the app.

Petra Miková
ID: 120852





Secondary criteria

6. Design patterns

In my implementation, I use the **Singleton design pattern** in classes UserManager - uses the Singleton design pattern to ensure that only one instance of the class can exist at any given time, and provides access to a single instance of the currentUser object of type T which extends the User class, and OrderManager - uses the Singleton design pattern to ensure that there is only one instance of the class at any given time, which provides the ability to work with one order object throughout the whole order creation even in different controllers.

Besides this design pattern, my implementation also uses the **MVC (Model-View-Controller) design pattern:**

Model: The objectclasses, tools, exceptions and app packages that contain the application logic and represent the data and logic of my application.

View: The FXML directory in resources folder with user interface (UI) files. These files describe the structure and appearance of the UI and how the UI interacts with the user.

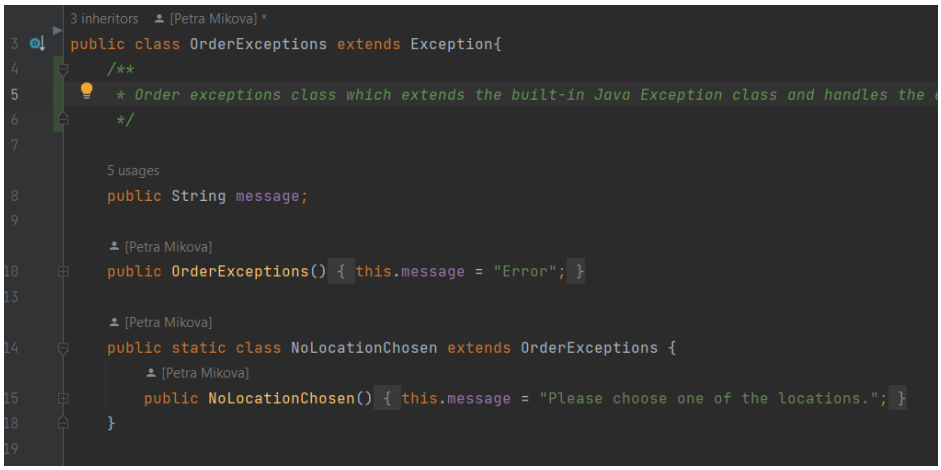
Controller: The classes in the controllers package handle user input, manage the state of the UI, and communicate with the model to update the data and logic.

7. Own exceptions

My implementation also includes handling exceptional states using own exceptions. For this, I have implemented two classes for handling exceptions during registration and login, and during the

order creation – RegistrationAndLoginExceptions and OrderExceptions. These inherit from the builtin Java Exception class and have several subclasses for specific exceptions which are then thrown in the application process.

An example of my own exception:



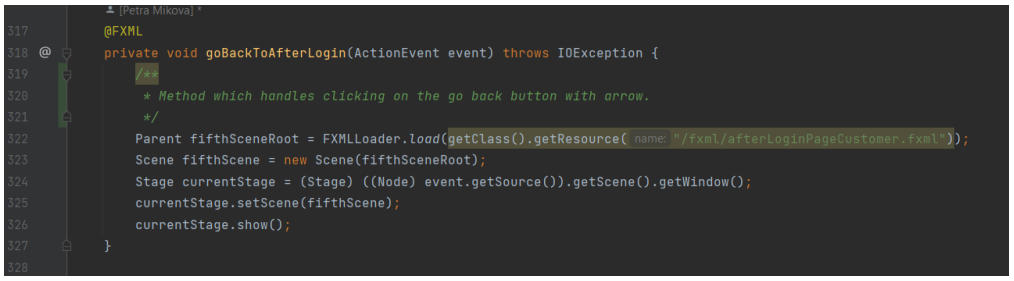
The message is then overriden like this:

8. Graphical user interface separated from application logic

My GUI is fully separated from the logic as I already mentioned. All the GUI .fxml files are in the resources directory and the interaction from user is handled via controllers using the FXML functions with parameter ActionEvent. The controllers are in the controllers package of my app.

Some of the examples:

1. goBackToAfterLogin in OrderController class

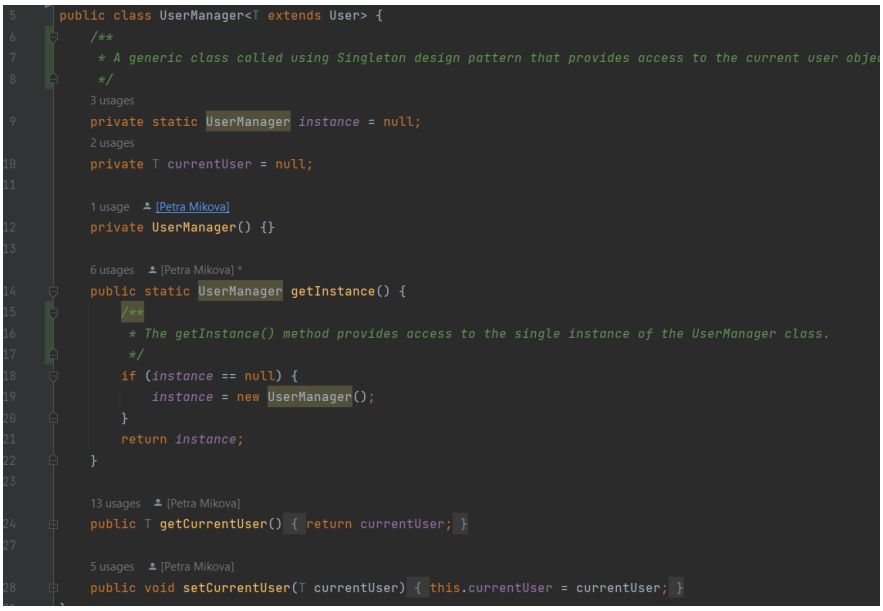


2. hyperLinkHandlerTripSummary in TripSummaryController

```
@FXML
public void hyperLinkHandlerTripSummary(ActionEvent event) throws IOException {
    /**
     * Method that when user clicks on their itinerary, all the places they have chosen in previous step will pop up.
     */
    List<Place> selectedPlacesHyper = customerUserManager.getCurrentUser().getPreferences().getSelectedPlaces();
    VBox mainBox = new VBox();

    for (Place place : selectedPlacesHyper) {
        VBox infoBox = new VBox();
        infoBox.getChildren().addAll(
                new Label( text: "Place: " + place.getName())
        );
        mainBox.getChildren().add(infoBox);
    }

    mainBox.setStyle("-fx-font-size: 16px;");
    Scene scene = new Scene(mainBox);

    Stage stage = new Stage();
    stage.setTitle("Places itinerary");
    stage.setScene(scene);
    stage.show();
}
```

9. Using generics in own classes

The generic type parameter T is declared in the class signature as UserManager<T extends User>, and it is used as the type of the currentUser field, which is declared as private T currentUser = null. The generic type T must extend the User class, which means that any object of the UserManager class can work with a specific subtype of User.



```
public class UserManager<T extends User> {
    /**
     * A generic class called using Singleton design pattern that provides access to the current user obje
     */
    3 usages
    private static UserManager instance = null;
    2 usages
    private T currentUser = null;

    1 usage  [Petra Mikova]
    private UserManager() {}

    6 usages  [Petra Mikova] *
    public static UserManager getInstance() {
        /**
         * The getInstance() method provides access to the single instance of the UserManager class.
         */
        if (instance == null) {
            instance = new UserManager();
        }
        return instance;
    }

    13 usages  [Petra Mikova]
    public T getCurrentUser() { return currentUser; }

    5 usages  [Petra Mikova]
    public void setCurrentUser(T currentUser) { this.currentUser = currentUser; }
```

10. Explicit use of RTTI

In the PlacesController class in method checkBoxesSetSelectedPlaces I use RTTI (instanceof) to determine the type of an object and retrieve chosen places by user in the GUI.

```java
public void checkBoxesSetSelectedPlaces(ActionEvent event) throws IOException {
    /**
     * Method that handles retrieving the chosen places and setting them to customer preferences.
     */
    Customer thiscustomer = customerUserManager.getCurrentUser();
    List<Place> selected = new ArrayList<>();
    for (Node node : checkboxContainerRestaurants.getChildren()) {
        if (node instanceof CheckBox) {
            CheckBox checkBox = (CheckBox) node;
            if (checkBox.isSelected()) {
                String id = checkBox.getId();
                for (Place place : allPlaces){
                    if (place.getId2().equals(id)){
                        selected.add(place);
                    }
                }
            }
        }
    }

    for (Node node : checkboxContainerSights.getChildren()) {
        if (node instanceof CheckBox) {
            CheckBox checkBox = (CheckBox) node;
            if (checkBox.isSelected()) {
                String id = checkBox.getId();
                for (Place place : allPlaces){
                    if (place.getId2().equals(id)){
                        selected.add(place);
                        pricesForPlaces += place.getPrice();
```

11. Using nested classes and interfaces

In my implementation, I use nested classes in both the exceptions classes – RegistrationAndLoginExceptions and Order exceptions. Both of these have several nested static classes which extend the main one and override displaying the message.

Example from my code:

```java
public class RegistrationAndLoginExceptions extends Exception {
    /**
     * Registration and login exceptions class which extends the built-in Java Exception class and handles the e
     */
    9 usages
    public String message;

    [Petra Mikova]
    public RegistrationAndLoginExceptions() { this.message = "Error"; }

    [Petra Mikova]
    public static class UsernameTakenException extends RegistrationAndLoginExceptions {
        [Petra Mikova]
        public UsernameTakenException() { this.message = "Username already taken."; }
    }

    [Petra Mikova]
    public static class BlankFieldsExceptionReg extends RegistrationAndLoginExceptions {
        [Petra Mikova]
        public BlankFieldsExceptionReg() { this.message = "Please fill in all the fields."; }
    }

    [Petra Mikova]
    public static class NoChoiceExceptionReg extends RegistrationAndLoginExceptions {
        [Petra Mikova]
        public NoChoiceExceptionReg() { this.message = "Please choose whether you are a customer or local."; }
    }
```

```java
public class OrderExceptions extends Exception{
    /**
     * Order exceptions class which extends the built-in Java Exception class and handles the excep
     */

    5 usages
    public String message;

    [Petra Mikova]
    public OrderExceptions() { this.message = "Error"; }

    [Petra Mikova]
    public static class NoLocationChosen extends OrderExceptions {
        [Petra Mikova]
        public NoLocationChosen() { this.message = "Please choose one of the locations."; }
    }

    [Petra Mikova] *
    public static class BlankFieldsExceptionOrder extends OrderExceptions {
        [Petra Mikova]
        public BlankFieldsExceptionOrder() { this.message = "Please fill in all the fields."; }
    }

    [Petra Mikova] *
    public static class InvalidNumberInput extends OrderExceptions {
        [Petra Mikova]
        public InvalidNumberInput() { this.message = "Please input valid number."; }
    }
```

I also use one interface in my implementation, called AfterLoginScreen, which declares basic methods that apply to both customer and local after they login into account. The classes OrderController and LocalOrderController implement this interface.

```java
2 implementations  new *
public interface AfterLoginScreen {
    2 usages  2 implementations  new *
    void LogOut(ActionEvent event) throws IOException;
    2 usages  2 implementations  new *
    void setHelloUser(String userFirstName);
}
```

12. Using lambda expressions or method references

I use lambda expressions in method ordersForReview in LocalOrderController class.

```java
98
99          Button acceptButton = new Button( text: "Accept");
100         acceptButton.setOnAction(actionEvent -> handleAcceptButtonClick(order, orderId, age, use
101         orderBox.getChildren().add(acceptButton);
102
103         Button rejectButton = new Button( text: "Reject");
104         rejectButton.setOnAction(actionEvent -> handleRejectButtonClick(order, orderId));
105         orderBox.getChildren().add(rejectButton);
106
```

13. Using serialization

In the User class, I use serialization to read and write user data to a binary file called "users.dat". In the getUsers() method, I create an ObjectInputStream that reads the data from the "users.dat" file and deserializes it into an ArrayList of User objects. Then I return this list of users. In

the setUsers() method, I take an ArrayList of User objects as an argument and create an ObjectOutputStream. Finally, I serialize the users ArrayList and write it to the "users.dat" file.

```
109     public static ArrayList<User> getUsers() {
110         /**
111          * This method reads user data from a file named "users.dat" using ObjectInputStream and returns an ArrayL
112          */
113         ArrayList<User> users = new ArrayList<>();
114         try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream( name: "users.dat"))) {
115             users = (ArrayList<User>) ois.readObject();
116         } catch (FileNotFoundException e) {
117         } catch (IOException | ClassNotFoundException e) {
118             e.printStackTrace();
119         }
120         return users;
121     }
122
            1 usage   [Petra Mikova] *
123     private static void setUsers(ArrayList<User> users) {
124         /**
125          * A private static method that takes an ArrayList of User objects as its argument. It serializes the Arra
126          */
127         try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream( name: "users.dat"))) {
128             oos.writeObject(users);
129         } catch (IOException e) {
130             e.printStackTrace();
131         }
132     }
133 }
```

Functionality of the app

A gif visually representing the functionality is in ReadMe file documentation on GitHub.

1. You can sign in – either as a local or customer. If you register as a local, you fill out further registration form. The users are serialized into users.dat file.
2. Then you log in – if you are customer, you have two buttons to create a new order or view your current trips and their status. If you are a local, you have two buttons to review orders and other to see accepted trips. In both cases you can log out there.
3. When creating new order as a customer, you are led through multiple scenes to choose your preferences for local and the trip itself. The places are stored in a JSON database which is deserialized into Places array list. In each step you have the option to go back. Eventually, you get to trip summary scene, where you can either reject or accept the offer. If you accept it, it is stored in a SQLite database and available for locals to review from now on.
4. Then you can simply see your current trips by clicking on the button in after login page.
5. As a local, if there are any trips for you to review, they will appear after clicking on the button made for reviewing offers. For every trip, you have accept and reject button so you can decide on each trip. If you accept any, they will pop up after clicking on a button made for it.

Petra Miková
ID: 120852

Main GitHub versions

Commits on Mar 20, 2023

- included gradle into project
- javafx plugin working

Commits on Mar 23, 2023

- essential classes added
- simple GUI

Commits on Apr 3, 2023

- working login and registration with functional GUI
- exceptions for login and registration

Commits on Apr 11, 2023

- Working Program Version
- order creation process, setting the preferences on Local
- redirection to page with places in chosen city, and provided description of every place
- redirection to trip summary page (was a WIP)

Commits on Apr 27, 2023

- SQLite for orders added
- modified build.gradle to be fully functional
- module info added

Commits on May 2, 2023

- functional SQLite communication and customer being able to accept or reject the order and thus change the status in the database

Commits on May 8, 2023

- interaction with order from local's side functional
- log out and go back to previous page buttons
- GUI finalised
- division of code into packages

Commits on May 13, 2023

- final commit
- fully completed application
- documentation + JavaDoc added into Documentation repo
- ReadMe finalised

Conclusion

A travel app, as it was stated in the general topic, covers the connection between people who want to travel to three locations in Spain and people from Spain who offer stay at their home.

It is provided with authentification of the user (login) and offers possibility for customer to create a trip exactly how they want, so that their preferences are fullfiled. Then it is up to review for local and if they accept it as well, the adventure can begin!

For storing the information, I used serialization into .dat file for login information, JSON database for beforehand stored places for each location, and SQLite database for storing orders and updating their status.

My assignment fulfilled the main criteria of OOP which were inheritance, aggregation, polymorphism, encapsulation, and division into packages. Besides that, it fullfiled many of the secondary criteria, which have been described earlier in this documentation. Thus, I would state that the app fulfills the assignment and subject requirements.