**Slovenská technická univerzita v Bratislave**

**Fakulta informatiky a informačných technológií**

**Ilkovičova 2, 842 16 Bratislava 4**

Predmet / Subject

**– Objektovo-orientované programovanie /**
**Object oriented programming –**

**- Dokumentácia / Documentation -**

# Hike de Slovakia

Ak. Rok / Academic term: 2022/2023, letný semester

**Cvičiaci / Instructors:**

Mgr. Pavle Dakič

**Študent / Student:**

Adam Grík

# Adam Grik - Hike de Slovakia

## Project Objective

### Slovak version

Aplikácia bude slúžiť používateľom, či už zo Slovenska alebo z celého sveta, ktorí budú chcieť navštíviť Slovensko za účelom turistiky. Hlavný cieľ aplikácie bude zostaviť čo najzaujímavejšiu trasu spoznávania, ktorá bude pozostávať z turistických trás na základe toho aké informácie používateľ zadal.

Aplikácia bude typu desktop aplikácie, a bude vytvorená v programovacom jazyku Java a JavaFX.

Aplikácia bude vedieť zostaviť tzv. turistickú trasu spoznávania na základe viacerých kritérií, ktoré bude používateľ zadávať .

Hlavným kritériom bude dĺžka spoznávania, to znamená, že používateľ si bude môcť vybrať koľko dní bude turistická trasa spoznávania trvať. Ako ďalšie si bude môcť používateľ zvoliť, v ktorej časti Slovenska (západ, stred, východ) bude chieť turistickú trasu spoznávania absolvovať. Následne si používateľ bude môcť tak isto zvoliť náročnosť turistických trás, ktoré mu aplikácia pridá do trasy spoznávania. Stupne obtiažnosti budú tri (nízka, stredná vysoká), to znamená že aplikácia pridá do trasy spoznávania také turistické trasy, ktoré sa zhodujú s danou obtiažnosťou, ktorú si používateľ zvolil.

Po procese vytvorenia trasy, bude môcť používateľ vytvorenú trasu nájsť vo svojom profile a pozrieť si jej detaily, to znamená že si bude môcť pozrieť detaily každej turistickej trasy, ktorá sa bude nachádzať v trase spoznávania.

Používateľ bude mať možnosť absolvovať vytvorenú trasu, to znamená že ak absolvuje nejakú turistickú trasu ktorá sa nachádza v danej trase spoznávania bude si ju môcť označit ako absolvovanú a bude môcť sledovať svoj progres, čo znamená, že ak absolvuje všetky turistické trasy bude celá trasa spoznávania označená ako absolvovaná.

### English version

The application will serve users, whether from Slovakia or from all over the world, who want to visit Slovakia for hiking purposes. The main goal of the application will be to compile the most interesting itinerary, which will consist of hiking routes based on the information entered by the user.

The application will be able to build a so-called hiking route based on several criteria that the user will enter.

The application will be of the desktop application type, and will be created in the Java and JavaFX programming languages.

The main criterion will be the length of the exploration, i.e. the user will be able to choose how many days the exploration route will take. Next, the user will be able to choose in which part of Slovakia (west, centre, east) he/she wants to follow the hiking route of discovery. Then the user will also be able to choose the difficulty of the hiking trails that the application will add to the discovery route. There will be three levels of difficulty (low, medium, high), which means that the application will add to the discovery route the hiking routes that match the difficulty chosen by the user.

After the route creation process, the user will be able to find the created route in his/her profile and view its details, i.e. he/she will be able to view the details of each hiking route that will be located in the discovery route.

The user will be able to absolve the created route, which means that if he/she completes a hiking route that is in a given discovery route, he/she will be able to mark it as absolved and he/she will be able to track his/her progress, which means that if he/she completes all the hiking routes, in the entire discovery route will be marked as completed.

## Table of Contents

- Project documentation
- Project documentation is also provided in JavaDoc in GitHub repositary

## Versions (All application functionality)

- Version 1.0.0:
  - **User registration** - The user will be able to register for the application by clicking on the register button, filling in all the necessary data and then having an account created in the application.
  - **User login** - If the user has already registered, he/she can log in to the application using the username and password he/she entered during registration.
  - **Journey creation** - Once a user has successfully logged in to the app, the user can create his or her tourist route of discovery based on whatever criteria he or she specifies.
  - **Display journey** - If a user creates a route, the name of every single route created will be displayed on the main page of the user's profile.
  - **Display places of journey** - On the main page, the user selects the route whose places they want to view. The user then clicks on the display button and the application displays a set of all the places that are in the route. He can then select the specific place whose information he wants to view.
  - **Display details of each place in created journey** - After the user selects which journey they want to view, they will be shown a detail of each place in that journey. It shows the name, description of the place, hiking route, total length, total duration, total elevation and also the maximum altitude reached.
  - **Marking place as visited** - In the place detail the app will also show two toggle buttons where the user can mark the route as either visited or

unvisited, once the user marks all the places as visited the journey will be automatically moved to completed journeys.

- **Display completed journey** - After the journey is automatically moved between the completed journeys, the user will be able to select that he wants to view the complete journey. After clicking on this button, the statistics of the journey will be displayed. Among the statistics will be displayed: total number of places visited, total number of kilometers, total number of vertical meters, total time. The user will then have 2 options, either to return to the main menu and keep the journey saved, or to delete the route completely from the system.

- Final version:
  - **Add notes for place** - If the user has created a journey, and is on a screen that displays the places of the journey, user can choose to add or display notes for a specific place on the journey. Then the user can add and view notes for that place, but can only add a maximum of 3 notes for each place. This way he can add notes to each place in each route which is created in his profile. The created notes can be viewed and read at any time
  - **User settings** - The user can select the user settings option in the main menu. After selecting and displaying this window, the user can change their password, or change their username, or delete an already created route.
  - **Admin user** - in final version was also implemented admin user.
    - **LOGIN CREDENTIALS FOR ADMIN - USERNAME: adminadmin, PASSWORD: adminadmin**
    - Admin can manage all accounts. After logging in, it sees a table of all users on the system. When admin click on a specific user, application will display information about that user such as username, password, first name, last name and also the time of the last login. In this window, the admin can also change the username of the user and delete all their journey, whether created or complete or delete whole username account.
    - Admin can also choose to add a new user. After clicking on this option, the admin will see a window with the fact that he must fill in all the registration information about the new user, if he fills in everything and the user with the username will not exist yet tehn the new account will be created.

**Data storage**

- Application data are stored outside the application logic. They are stored using serialization, where every time the data is changed, whether it is adding a user, deleting a user, creating a route, completing a route, marking a visited place, adding notes for individual places on the route, all this is written to a file using the serialization, where then after turning off and on again the application all this data will be reloaded again and will not be lost.
- For more information on how serialization works, see the secondary criteria section.

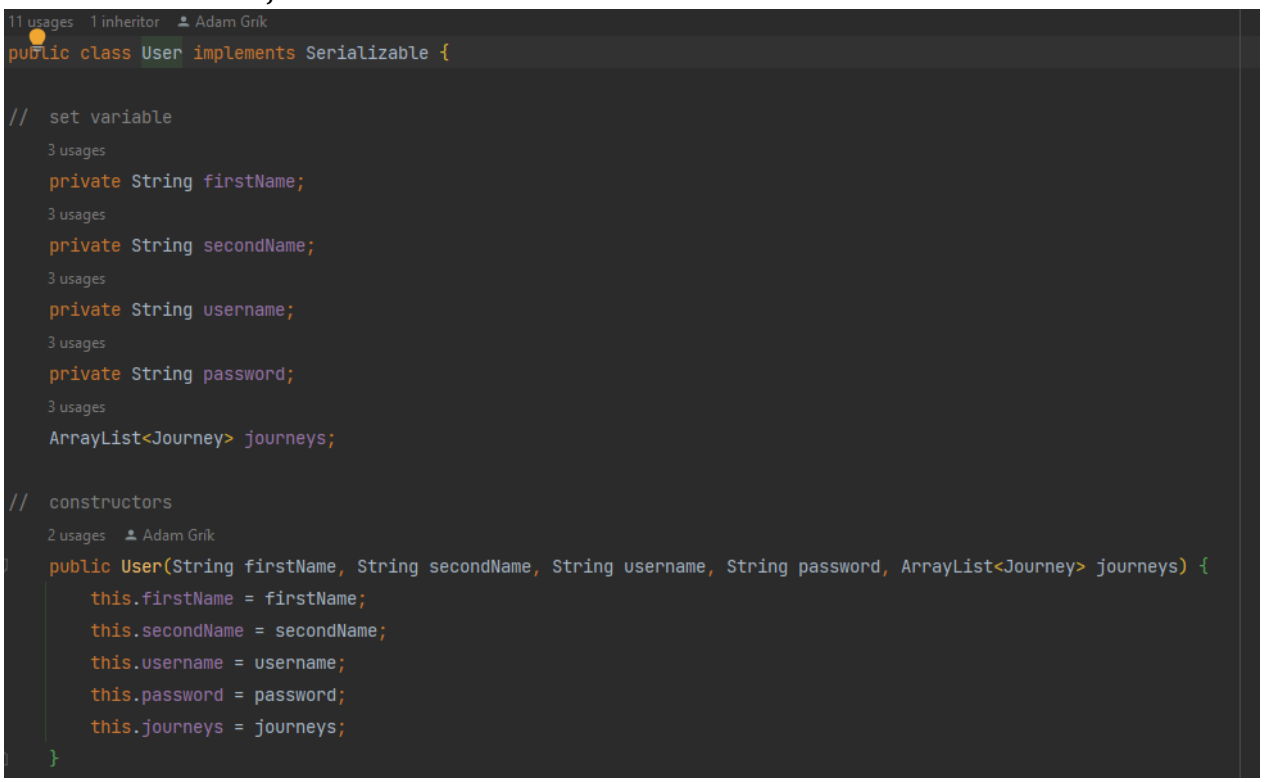
## Fulfillment of criteria

- Main criteria
  - **inheritance**
  - **aggregation**
  - **polymorphism**
- Secondary criteria
  - **serialization**
  - **Observer**
  - **generic class**
  - **nested class**
  - **providing a graphical user interface separate from the application logic**
  - **method references**
  - **default method implementation**
  - **custom exception**

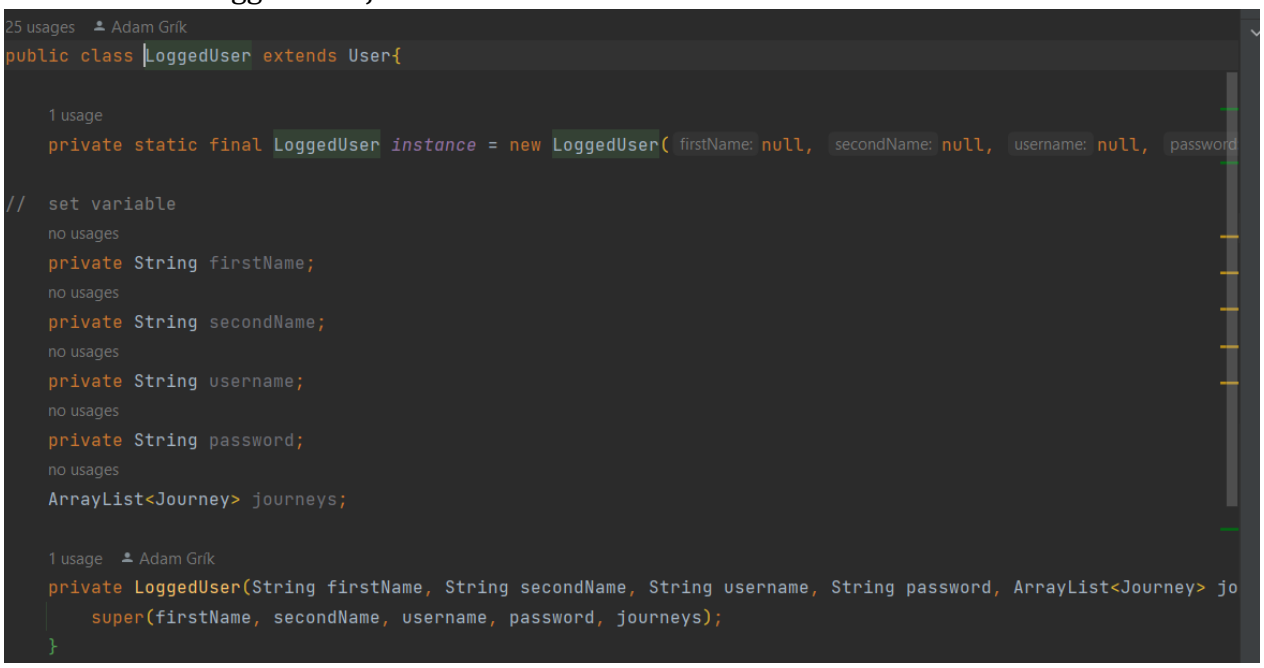
## Certain implementations

### Main criteria

- **inheritance** - is between classes User and LoggedUser, because LoggedUser can inherit variables and methods from User
  - User.java



- LoggedUser.java



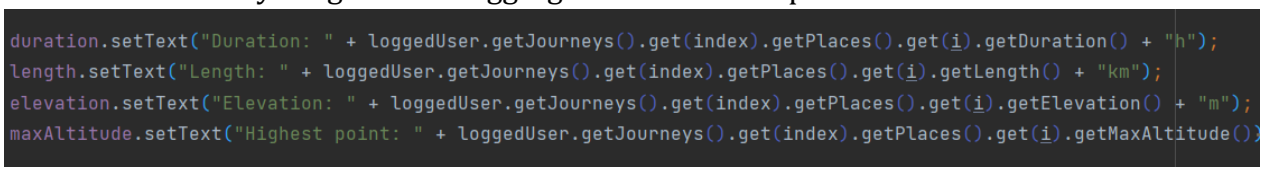
```
25 usages  Adam Grík
public class LoggedUser extends User{

    1 usage
    private static final LoggedUser instance = new LoggedUser( firstName: null,  secondName: null,  username: null,  password

//  set variable
    no usages
    private String firstName;
    no usages
    private String secondName;
    no usages
    private String username;
    no usages
    private String password;
    no usages
    ArrayList<Journey> journeys;

    1 usage  Adam Grík
    private LoggedUser(String firstName, String secondName, String username, String password, ArrayList<Journey> jo
        super(firstName, secondName, username, password, journeys);
    }
```

- **aggregation** - is between classes User and Journey, because we can't work with Journey class except through the User class
  - User.java
  - Journey.java
  - Usage - DisplayJourneyController.java from line 126 or 163 or 185, there are many usages of this aggregation relationship



```
duration.setText("Duration: " + loggedUser.getJourneys().get(index).getPlaces().get(i).getDuration() + "h");
length.setText("Length: " + loggedUser.getJourneys().get(index).getPlaces().get(i).getLength() + "km");
elevation.setText("Elevation: " + loggedUser.getJourneys().get(index).getPlaces().get(i).getElevation() + "m");
maxAltitude.setText("Highest point: " + loggedUser.getJourneys().get(index).getPlaces().get(i).getMaxAltitude()
```

- **polymorphism** - parent class is class GoBack which has abstract method *clickOnBack* which ensures that if the user clicks the back button, hi will be redirected to the previous screen
  - child classes -
    - AddNoteController.java, this class inherits from the GoBack class and uses its abstract method 111

```
@Override
public void clickOnBack(ActionEvent event) throws IOException {
    switchScene(event, newScene: "/views/displayJourneyPage.fxml");
}
```

    - UserDetailsController.java, this class also inherits from the GoBack class and uses its abtract method 322

```
@Override
public void clickOnBack(ActionEvent event) throws IOException {
    switchScene(event, newScene: "/views/adminHomePage.fxml");
}
```

  - we can see that both of these classes inherit and use the same method from the GoBack class, but in each of these classes the clickOnBack method is referenced to a different screen, so the methods behave differently. But the method *clickOnBack* is used in many other classes

**Secondary criteria**

- **Observer** - in DisplayJourneyController.java from line 123 where the Toggles are observed as the Observable Value, if user click on them or not, the value which represents visiting of place is changed on the basis of clicking on the Toggles

```
toggleGroup.getToggles().addAll(unvisitedButton, visitedButton);

toggleGroup.selectedToggleProperty().addListener((ov, toggle, newToggle) -> {
    int placeIndex1 = 0;
    for(int i=0; i < loggedUser.getJourneys().get(index).getPlaces().size(); i++) {
        if(loggedUser.getJourneys().get(index).getPlaces().get(i).getName().equals(name.getText())
            placeIndex1 = i;
            break;
        }
    }
```

- **generic class** - is in class StringHolder, where the parameter of class is String because class StringHolder holds string value, between two .fxml views
  - StringHolder.java

```
toggleGroup.getToggles().addAll(unvisitedButton, visitedButton);

toggleGroup.selectedToggleProperty().addListener((ov, toggle, newToggle) -> {
    int placeIndex1 = 0;
    for(int i=0; i < loggedUser.getJourneys().get(index).getPlaces().size(); i++) {
        if(loggedUser.getJourneys().get(index).getPlaces().get(i).getName().equals(name.getText())
            placeIndex1 = i;
            break;
        }
    }
```

  - Usage - MainPageController.java from line 82

```
1 usage   Adam Grík
public void clickOnDisplayCompletedJourney(ActionEvent event) throws IOException {
    if(completedJourneys.getValue() != null) {
        StringHolder<String> stringHolder = StringHolder.getInstance();
        stringHolder.setObj(completedJourneys.getValue());
        switchScene(event, newScene: "/views/displayCompletedJourney.fxml");
    }
```

- **nested class** - is in class Journey and the nested class is Place, because the Place class is a direct part of the Journey class, and the Place class cannot be accessed in any other way than through the Journey class, so in this case it was better to use a nested class
  - Journey.java

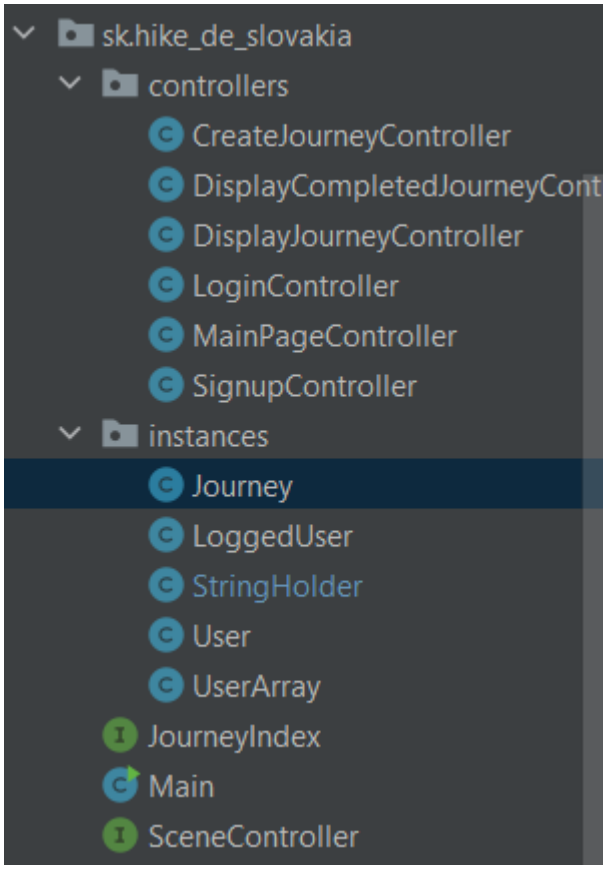
```
16 usages
public class Journey {
//    variables
    3 usages
    String name;
    3 usages
    ArrayList<Place> places;
    3 usages
    boolean isCompleted;

//    constructor
    3 usages
    public Journey(String name, ArrayList<Place> places, boolean isCompleted) {
        this.name = name;
        this.places = places;
        this.isCompleted = isCompleted;
    }
```

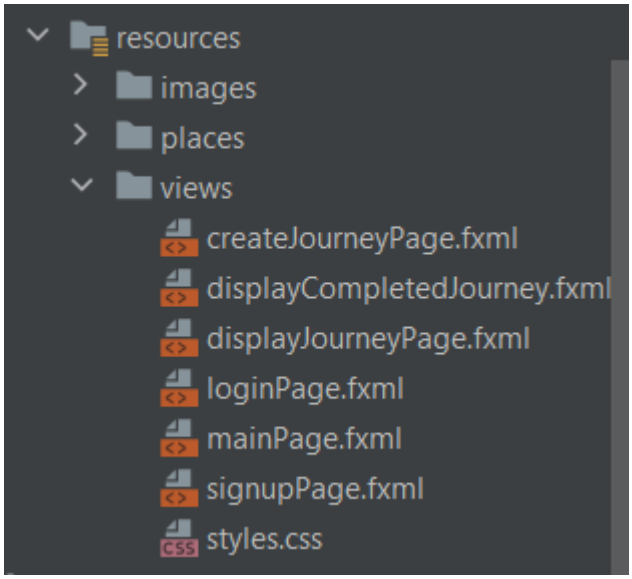
- Place.java from line 63

```
8 usages   Adam Grík
public static class Place {
    //    variables
    3 usages
    String name;
    3 usages
    String shortName;
    3 usages
    String desc;
    3 usages
    String hikeRoute;
    3 usages
    String duration;
    3 usages
    String length;
    3 usages
```

- **providing a graphical user interface separate from the application logic**
  - Here we can see that I have split instances and controllers into packages



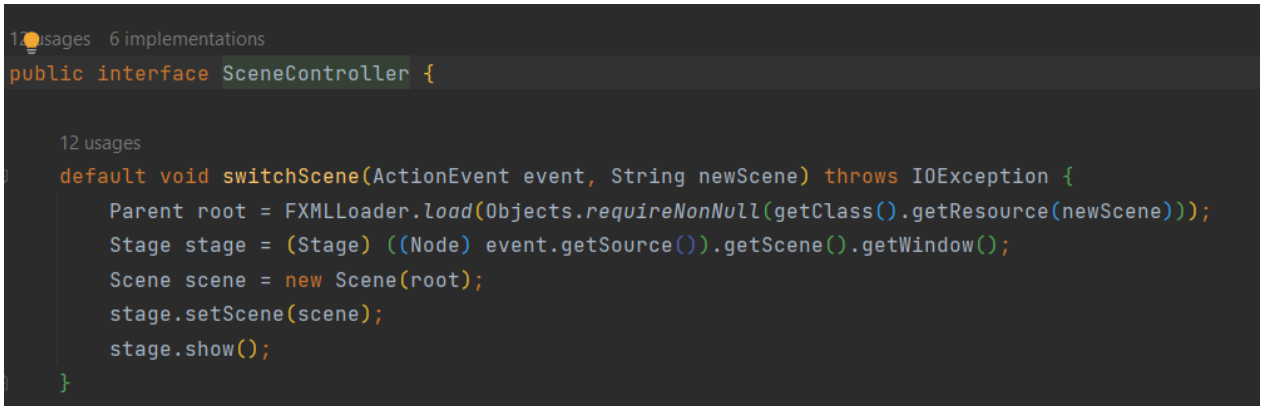
- Here we can see that I have also split views package and have it in separate package




- **method references** - is in DisplayJourneyController.java from line 123, where the one method references on the second method, which detects that which Toggle is clicked and the based on that does important stuff

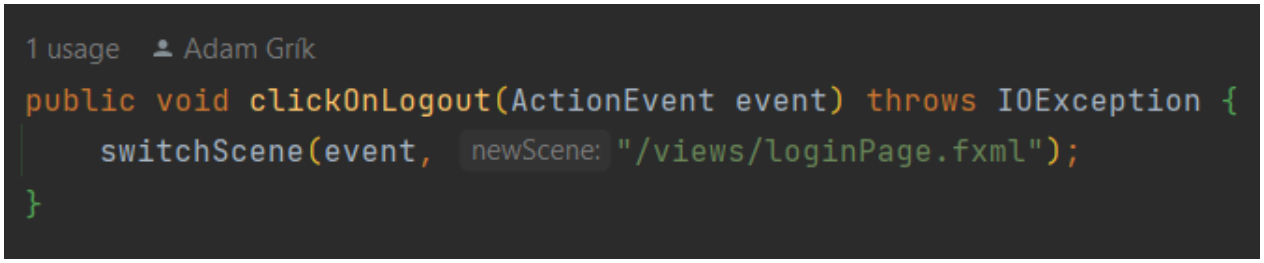
```
toggleGroup.selectedToggleProperty().addListener((ov, toggle, newToggle) -> {
    int placeIndex1 = 0;
    for(int i=0; i < loggedUser.getJourneys().get(index).getPlaces().size(); i++) {
        if(loggedUser.getJourneys().get(index).getPlaces().get(i).getName().equals(name.getText())) {
            placeIndex1 = i;
            break;
        }
    }
```

- **default method implementation** - is in interface SceneController and JourneyIndex
  - SceneController.java - in this interface is method switchScene which provides switching between two .fxml views, this method is used in every controller class in the same way, and therefore must be implemented as default method, usage for example here in class MainPageController on line 73 or 77



```
12 usages  6 implementations
public interface SceneController {

    12 usages
    default void switchScene(ActionEvent event, String newScene) throws IOException {
        Parent root = FXMLLoader.load(Objects.requireNonNull(getClass().getResource(newScene)));
        Stage stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.show();
    }
```



```
1 usage   Adam Grík
public void clickOnLogout(ActionEvent event) throws IOException {
    switchScene(event,  newScene: "/views/loginPage.fxml");
}
```

  - JourneyIndex.java - in this interface is method getIndexOfJourney, which provides finding the index of the given journey that we need to display, and also must be implemented as default method because is used in many controller class, for example in class DisplayJourneyController in line 225 or 231

```
4 usages  2 implementations
public interface JourneyIndex {

    4 usages
    public default int getIndexOfJourney() {
        StringHolder<String> stringHolder = StringHolder.getInstance();
        LoggedUser loggedUser = LoggedUser.getInstance();
        int index = 0;

        for(int i=0; i < loggedUser.getJourneys().size(); i++) {
            if(loggedUser.getJourneys().get(i).getName().equals(stringHolder.getObj())) {
                index = i;
                break;
            }
        }
        return index;
    }
}
```

- **serialization** - serialization is used in my application for all data that is stored even after shutting down the application. It is all data such as created users, created user routes, information about whether a route is completed or not, also saved for example notes for places, and much more.
  - Serializable classes: User, Journey and Place
  - Usage - serialization is used in all of the controllers, here is few examples:
    - SignupController.java from line 88, where when a user is registered, first an array of all users is loaded from the serialization file, and then the newly created user is added to this array and this array is written to the file again

```
ArrayList<User> users = null;

try (FileInputStream fis = new FileInputStream( name: "userData");
     ObjectInputStream ois = new ObjectInputStream(fis);) {

    users = (ArrayList) ois.readObject();

} catch (IOException ioe) {
//      ioe.printStackTrace();
} catch (ClassNotFoundException c) {
    System.out.println("Class not found");
//      c.printStackTrace();
}

if(users == null ) {
    users = new ArrayList<>();
}

users.add(user);
```

```
try (FileOutputStream fos = new FileOutputStream( name: "userData");
     ObjectOutputStream oos = new ObjectOutputStream(fos);) {

    oos.writeObject(users);

} catch (FileNotFoundException e) {
    throw new RuntimeException(e);
} catch (IOException ioe) {
    ioe.printStackTrace();
}
```

- AddNoteController.java from line 97 in method addNoteIntoFile, in this case we first load the array of all users from the file, then we find a specific user and his specific path that is currently open, then we add a note to the place for which the user has written a note, and again we write this array of users to the file, which ensures that the note for a specific place will be saved and loaded even after shutting down the application

```
1 usage   Adam Grik
public void addNoteIntoFile(LoggedUser loggedUser, String note, int index, int i) {
    ArrayList<User> users = null;
    try (FileInputStream fis = new FileInputStream( name: "userData");
         ObjectInputStream ois = new ObjectInputStream(fis);) {

        users = (ArrayList) ois.readObject();

    } catch (IOException ignored) {}
    catch (ClassNotFoundException c) {
        System.out.println("Class not found");
    }

    for(int j = 0; j < Objects.requireNonNull(users).size(); j++) {
        if(users.get(j).getUsername().equals(loggedUser.getUsername())) {
            users.get(j).getJourneys().get(index).getPlaces().get(i).addNote(note);
        }
    }

    try (FileOutputStream fos = new FileOutputStream( name: "userData");
         ObjectOutputStream oos = new ObjectOutputStream(fos);) {

        oos.writeObject(users);

    } catch (FileNotFoundException e) {
        throw new RuntimeException(e);
    } catch (IOException ioe) {
        ioe.printStackTrace();
    }
```

- **custom exception** - custom exception is class EmptyFieldsException

```
public class EmptyFieldsException extends Exception {

    public EmptyFieldsException() { super("Field username or password cannot be empty"); }
}
```

  - Usage - my custom exception is used in class LoginController on line 50, where during user login application check if both username and password fields are filled in, if one of them is not my own exception is called

```
1 usage   Adam Grik *
public boolean checkEmptyTextFields() {
    boolean isEmpty = false;
    try {
        if (textfield_username.getText().isEmpty() || passwordfield_password.getText().isEmpty()) {
            isEmpty = true;
            throw new EmptyFieldsException();
        }
    }
    catch (EmptyFieldsException e) {
        wrong_login.setText("");
        fields_empty.setText(e.getMessage());
    }
    return isEmpty;
}
```

## Class diagram