Lab 07: MVC Pattern

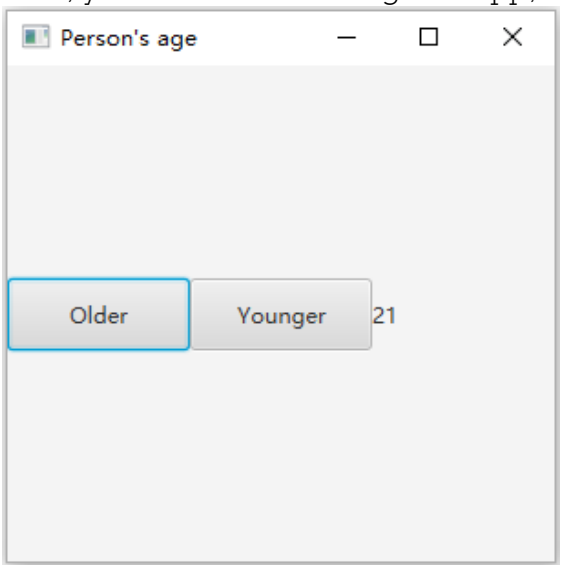
Aims:

1. Understand and practice the MVC pattern with samples.

MVC pattern, a sample implementation

Let's implement a sample MVC patterned project. Assume that your Model is a **Person** class containing an int typed field, named '*age*'. It provides the functionality of manipulating the age by incrementing or decrementing it.

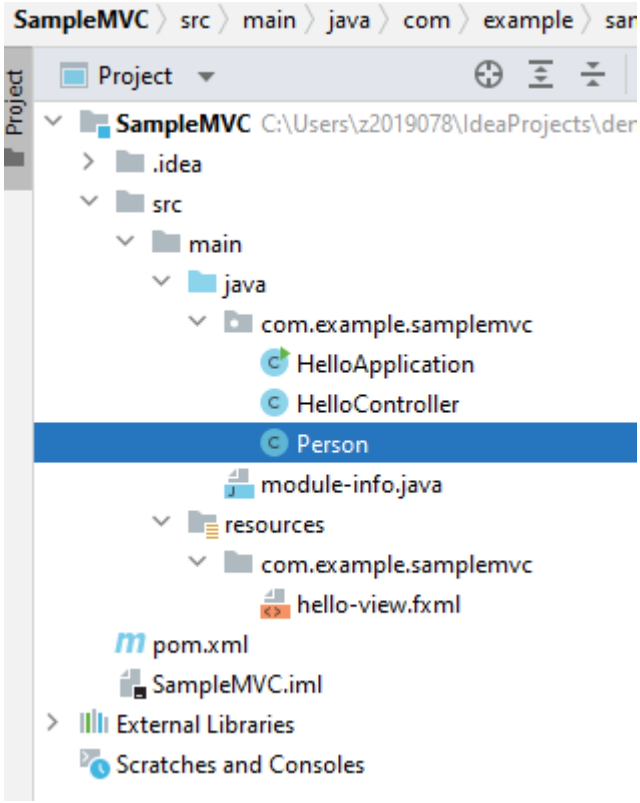
Now, you are asked to design an App, which looks like below (too simple to be true):



With this App, if the user clicks the 'Older' button once, the age is incremented by 1, and if the user clicks the 'Younger' button once, the age is decremented by 1. Of course, the displayed age should be linked to the **Person** at the back. More specifically, the buttons and label (i.e. the area that display the age number) *UIs* should be linked to the *model* Person, via some kind of *controller*.

Let's implement it using the MVC pattern. Create a JavaFX project in IntelliJ as instructed in last week's lab, named 'SampleMVC'.

In the 'samplemvc' package, create a **Person** class that includes an *integer* field named '*age*'. This is the Model.



- Note that its type should be defined as **IntegerProperty** instead of int, since this is a value that can invoke event listening. Revisit the last week's lab manual 06, to understand the usage of **Property**.

Create the constructor to instantiate the age object.

- Also note that when generating the getter and setter (by right click the word ***age***, and select 'generate…'), the IDE will also automatically generate an '***ageProperty()***' method, as shown below.

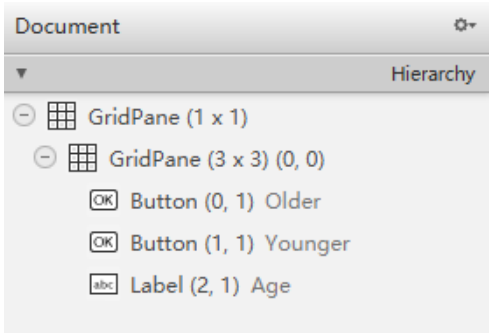
```
12  public class Person {
13      private IntegerProperty age;
14
15      public Person(int age){
16          if (this.age == null)
17              this.age = new SimpleIntegerProperty();
18
19          this.age.set(age);
20      }
21
22      public IntegerProperty ageProperty() {
23          return age;
24      }
25
26      public int getAge() { return age.get(); }
29
30      public void setAge(int age) { this.age.set(age); }
33
34      public void incrementAge() { this.age.set(getAge()+1); }
37      public void decrementAge() { this.age.set(getAge()-1); }
40  }
```

Also, in the Person model, add two additional methods to increment or decrement the age value.

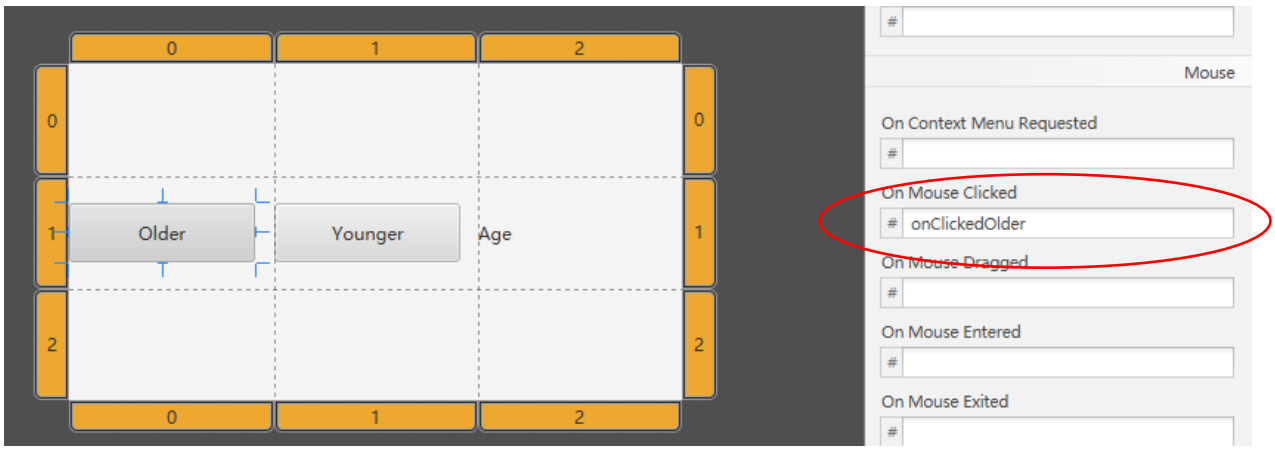
```
public void incrementAge() {
    this.age.set(getAge()+1);
}
public void decrementAge() {
    this.age.set(getAge()-1);
}
```

So this makes up the Model part. Now, let's design a View for the Model to interact with the user. This means the user will be able to manipulate the 'age' property in the Person class, by manipulating the UI component, such as clicking a button. Right click the 'hello-view.fxml' file, and select 'Open in SceneBuilder'.

In the SceneBuilder, drag and drop two buttons and a label onto a pane, as below. You can of course make a nicer looked version. After that, remember to save the work to update the 'hello-view.fxml' in the IntelliJ project.



Select the 'Older' button, and fill a method name 'onClickedOlder' on the event "On Mouse Clicked", then save.



In the 'hello-view.fxml', a line should appear to specify that the Button with text 'Older' is associated with a control method called "onClickedOlder", in the event "onMouseClicked", as below. But this method is currently not available in the Controller. Use the error prompt shown below to create a method '*onClickedOlder*' in the **HelloController** class definition.

Also be careful that you need to specify which controller should the "onClickedOlder" method be defined in 'hello-view.fxml':

```
fx:controller="com.example.samplemvc.HelloController"
```

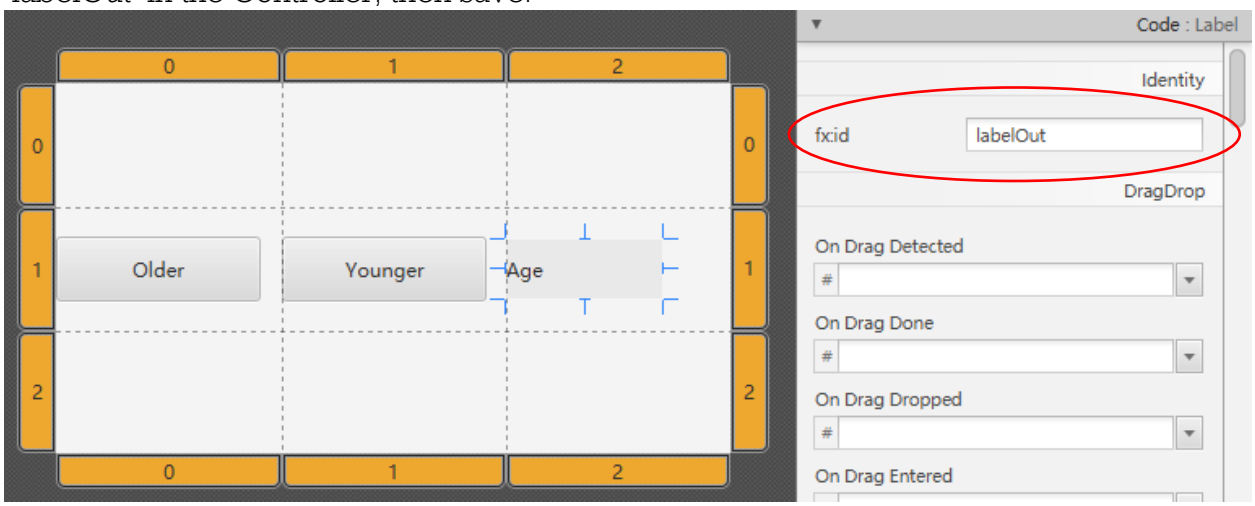
This is because you may define multiple Controller classes to control, e.g., various scenes. So it would be good to specify which UI component is controlled by which controller. In this example, the controller 'HelloController' is used to contain the action methods defined by the GridPane and the Buttons inside.

The signature of the "onClickedOlder" method is defined below.

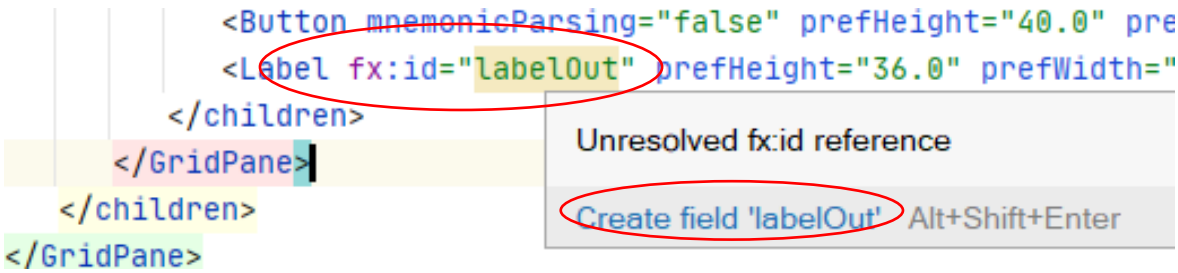
```
public void onClickedOlder(MouseEvent mouseEvent) {}
```

We will fill in the content of this method later.

Open the SceneBuilder again, and selection the 'Age' label, and give it a field name, 'labelOut' in the Controller, then save.



In 'hello-view.fxml', below information shows up



Choose to create the field 'labelOut', then in the **HelloController** class, a field named 'labelOut' appears. Add a @FXML annotation to this variable, so that this item will be automatically initialized by JavaFX using the information provided in 'hello-view.fxml'.

```
    @FXML private Label labelOut;
```

Also in the HelloController class, add an '*initialize()*' method annotated with @FXML, where JavaFX will execute initialization code.

Now, your HelloController class should look like the following.

```
9   public class Controller {
10
11      @FXML private Label labelOut;
12
13      public void onClickedOlder(MouseEvent mouseEvent) {
14
15      }
16
17      @FXML
18      public void initialize() {
19
20      }
21  }
```

Since the Controller's job is to glue the Model and View together, the Person class must be inside. We now add the Person class into the controller, as follows.

- A private variable '`person:Person`' is added (line 12).
- The `person` object is instantiated in the initialization method (line 20).
- When the button Older is clicked, the person's age should be incremented (line 15).
- The incremented age should be displayed in the label area, so the *ageProperty* is attached to a listener, which monitors the change of the age value. If any change happens, the label changes and displays its value (lines 21-30).

```
public class Controller {

    @FXML private Label labelOut;
    private Person person;

    public void onClickedOlder(MouseEvent mouseEvent) {
        person.incrementAge();
    }

    @FXML
    public void initialize() {
        person = new Person( age: 20);
        // add a listener
        person.ageProperty().addListener(
                new ChangeListener<Number>() {
                    @Override
                    public void changed(ObservableValue<? extends Number> observable,
                                        Number oldValue, Number newValue) {
                        labelOut.setText(newValue.toString());
                    }
                }
        );
    }
}
```

The HelloApplication class remains as below. In this simple case it is serving as the View.

```
public class HelloApplication extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception{
        Parent root = FXMLLoader.load(getClass().getResource( name: "hello-view.fxml"));
        primaryStage.setTitle("Person's age");
        primaryStage.setScene(new Scene(root,  v: 300,  v1: 275));
        primaryStage.show();
    }

    public static void main(String[] args) { launch(args); }
}
```

So, now we've realized a very simple MVC pattern. As you can see,

- The **Person** class is the barebone logic, like the Zoo example we've seen before. It must not have any view-specific elements inside.
- The **View** is in the HelloApplication class and realized with the help of the fxml mechanism, and is responsible of showing the content to the user. It is done with the help of the 'fxml' file via FXMLLoader. However, you can hand code the GUI, but the code would be messy for large projects.
- The **Controller** receives the input from the user (through event handling and listening), and invokes Person to make internal calculations.

Run it to see if it works.

This is still an over-simplified example, though. Also gain insights from a more comprehensive description:
https://docs.oracle.com/javafx/2/best_practices/jfxpub-best_practices.htm
https://stackoverflow.com/questions/32342864/applying-mvc-with-javafx

===
Task:
Complete the 'Younger' functionality of this example.
===

Sample code for this section is available on Moodle, named 'SampleMVC.zip'.