COMP 1406Z
Fall 2022 - Tutorial #4

Objectives

- Learn how to split a GUI into model, view and controller.
- To understand how to write proper coding style using an update() method.
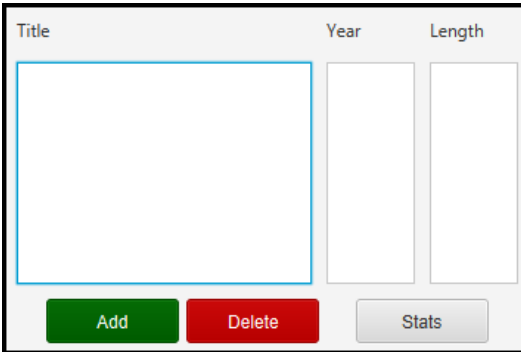- To become more familiar with event handling.

Getting Started:

Download the **Tutorial 4.zip** file from Brightspace. This file contains an IntelliJ project with the starting resources for this tutorial. You may have to follow the installation instructions for JavaFX posted on Brightspace in order to get the code to compile/run.
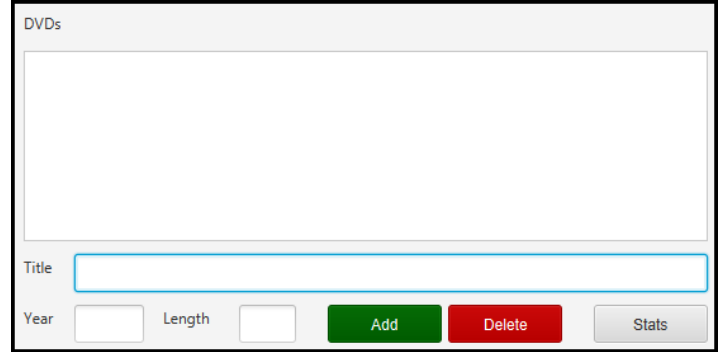
Tutorial Problems:

1) The tutorial code contains two different **Pane** classes that represent two different *views* of a **DVDCollection**.

These classes are called **DVDCollectionAppView1** and **DVDCollectionAppView2**:

DVDCollectionAppView1

DVDCollectionAppView2

There is also an application called **DVDCollectionApp1** which is used as the *controller* for the application. That is, it represents the main application onto which the view is attached.

    A. Run the **DVDCollectionApp1** class and ensure that the window contains the first view above.

    B. Change the line that creates the view to create a **DVDCollectionAppView2** view instead of a **DVDCollectionAppView1** view.

    C. Re-run the code and ensure that it looks like the 2nd view above.

Note that we will keep changing which view we have in the application throughout the tutorial.

    D. Rewrite the line so that it creates a **DVDCollectionAppView1** view again.

Notice how we can simply swap various views in and out of our application with this simple one-line change. This is one advantage of separating the *view* from the *controller* in our MVC strategy.

---

2) Now we will insert the *model* into the mix as our third part of the MVC. The model is a **DVDCollection** object.

A. Create an instance variable called **model** in the application class (i.e., not the view) to store a **DVDCollection** instance.

B. Add code to the blank constructor in the application class to set the model to **DVDCollection.*example1*();**

At this point, the *model* is now created and stored. The example *model* contains 5 DVDs. However, when we start the application, we still don't see the DVDs appearing. That is because we have not yet written code to update the contents of the three lists. Regardless of the *view* that we choose to use, all of them must be able to update their appearance to reflect the current state of the *model*.

To ensure that all views have an **update()** method, we will make them implement a common interface.

C. Create a new interface called **DVDView** as shown below:

```
public interface DVDView {

    // Cause view to update its appearance based on given model &
selected DVD

    public void update(DVDCollection model, int selectedDVD);

}
```

D. Now go into each of the two *view* classes and have them *implement* the **DVDView** interface by typing this method in each class:

```
    public void update(DVDCollection model, int selectedDVD) {


    }
```

Our code should now compile, but we still don't see the DVDs when we run. That is because we need to write code in the **update()** method.

E. Write the code in the **update()** method for **DVDCollectionAppView1**. It should create three arrays that will be used to fill in the three corresponding separate **ListViews (displaying title, year, and length of DVD).**

You will need to *get each of the three data elements* by accessing each object in the list of **DVD** objects. The list of **DVD** objects can be accessed through the *model* by using getDVDList() method.

F. Call the view's update method – view.update(model, 0); – from the start method in the **DVDCollectionApp1** class.

3

Make sure you add this line to the top of the class to avoid getting errors:

```
import javafx.collections.FXCollections;
```

G. Run the code and make sure that it looks like this:
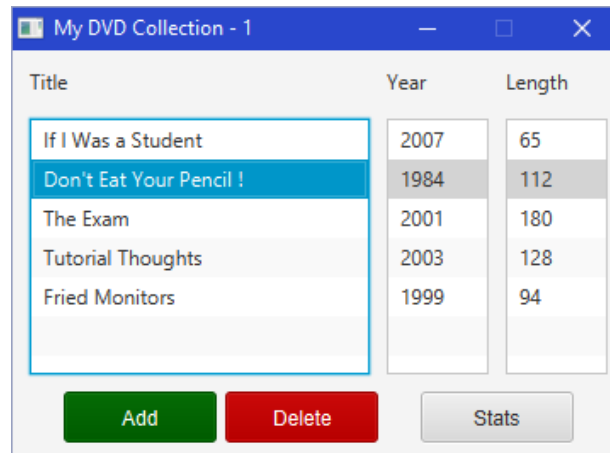
3) Write the code in the **update()** methods for the **DVDCollectionAppView2** class.

A. The code should get the list of **DVD** objects from the *model* and create the appropriate contents for the single **ListView**. Specifically, this *view* will contain **DVD** objects in the **ListView**.

B. Make sure you add this line to the top of the **DVDCollectionAppView2** class:

```
import javafx.collections.FXCollections;
```

C. Modify the app class to use the second view, run the code and make sure that it looks the same as the window below:
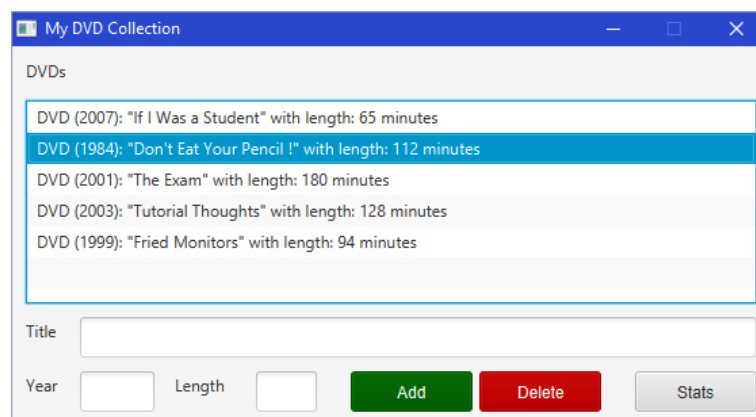
4) Make sure that the **DVDCollectionApp1** code is making use of the **DVDCollectionAppView1** view.

    A. Add this line to the top of the **DVDCollectionApp1** class.

```
import javafx.event.*;
```

    B. Create an event handler for the **Add** button in the **DVDCollectionApp1** class:

```
view.getButtonPane().getAddButton().setOnAction(new
EventHandler<ActionEvent>() {

        public void handle(ActionEvent actionEvent) {

        }

});
```

    C. You will need to fill in the code so that it asks the user for the DVD title, year and length. To do this, you will have to retrieve three pieces of information from the user – the title (String), the year (int) and the length (int). Here is a line of code that allows you to get a String from the user (**Note: in some cases, the code below does not work on Mac computers. If that is the case, you can use a basic Scanner to read the user's input from the console**):

```
String s = javax.swing.JOptionPane.showInputDialog("Please enter your
name: ");
```

The above code brings up a window (see below), waits for the user to respond, and then stores the String that was input into the variable **s**.
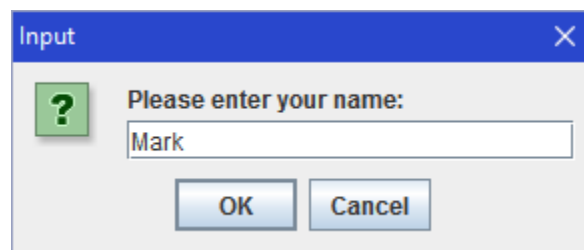
You will need to do something similar to get the title, year and length. Note that for the year and length, you will need to convert them to integers by using: **Integer.parseInt(s);**

    D. Once you get the information, you will need to:

        a. Modify the *model* by adding the **DVD** to the **DVDCollection**
        b. Update the *view*.

    E. Run the code and make sure that the DVD gets added and that the GUI gets refreshed to show the newly added **DVD** in the list. Note that your **update()** method

should already do all the hard work here. Do not re-write code to add anything to the **ListView**.

5) Similarly, create an event handler to handle the **Delete** button. It should delete the **DVD** whose title is selected in the title **ListView**.

You should follow the same strategy - change the *model* and then update the *view*.

Test the code to make sure that you can add and delete DVDs.

6) Adjust the code in **DVDCollectionApp1** by adding an event handler so that whenever the user clicks in the **Title** list, the three **ListViews** stay synchronized. That is, if the user selects the third item in the **Title** list, then the third item in the **Year** list and the third item in the **Length** list should automatically be selected.

This can be done easily through the **update()** method. You can handle selections easily in a **ListView** by using a **mousePressed** event handler:

```java
view.getTitleList().setOnMousePressed(new EventHandler<MouseEvent>()
{
    public void handle(MouseEvent mouseEvent) {

    }

});
```

Submission

Zip your completed tutorial #4 project and submit your **tutorial4.zip** file to the Tutorial #4 submission on Brightspace. Make sure you download and test your submission after you have submitted. Submitting a corrupt zip or a zip file that does not have the correct files will result in a loss of marks.