Object-Oriented Programming

LECTURER: NGUYEN THI THU TRANG, TRANGNTT@SOICT.HUST.EDU.VN
TEACHING ASSISTANTS: NGUYEN T.T. GIANG, GIANG.NTTI94750@SIS.HUST.EDU.VN
VUONG DINH AN, AN.VDI800003@SIS.HUST.EDU.VN

Lab 08: GUI Programming with JavaFX and Exception Handling

In this lab, you will practice with:

- Use SceneBuilder to design a graphical user interface
- JavaFX containers: AnchorPane, BorderPane, GridPane, HBox, VBox, …
- JavaFX data-driven UI: TableView, ListView, …
- JavaFX property binding
- Switch scenes in JavaFX application

0 Assignment Submission

For this lab class, you will have to turn in your work twice, specifically:

- **Right after the class**: for this deadline, you should include any work you have done within the lab class.
- **10 PM 7 days after the class**: for this deadline, you should include your work on all sections of this lab, and push it to a branch called "***release/lab08***" of the valid repository.

After completing all the exercises in the lab, you have to update the use case diagram and the class diagram of the AIMS project.

Each student is expected to turn in his or her work and not give or receive unpermitted aid. Otherwise, we would apply extreme methods for measurement to prevent cheating. Please write down answers for all questions into a text file named "**answers.txt"** and submit it within your repository.

1 Setup JavaFX for Eclipse

Note: This instruction is for JDK versions after 1.8. If you are using JDK 1.8, you can skip this installation since JavaFX is already integrated into JDK 1.8.

1.1 Install plugin for JavaFX

To work with JavaFX in Eclipse, you need to install some plugins. The most popular one is **e(fx)clipse**.

- Open Eclipse, on the Menu bar, choose Help → Eclipse MarketPlace → search **e(fx)clipse** and click Install.
- Follow the instruction and restart Workspace.

- Check after installation: After successful installation and restarting Eclipse, you can check the result of the installation. In Eclipse select: **File/New/Others**... There are Wizards which allow you to carry out JavaFX programming

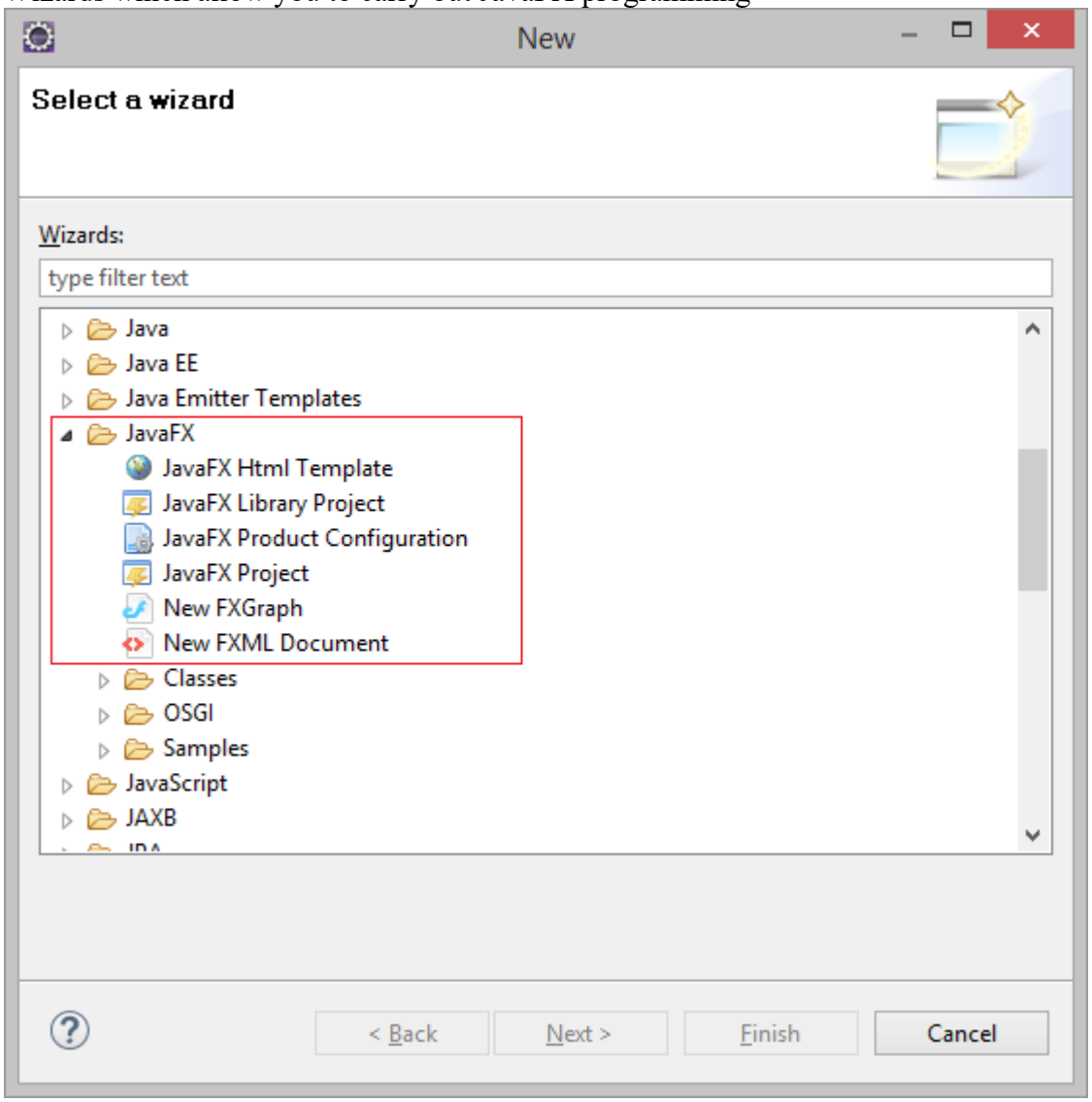


Figure 1. Check result after installing plugin

1.2 Download JavaFX

Go to this page https://gluonhq.com/products/javafx/, select the latest JavaFX version then select your Operating System and Architecture, for the Type option select SDK.
Download the JavaFX and extract it to a folder (please remember the location).

1.3 Add JavaFX to Eclipse

In this part, you will create a user library for JavaFX in Eclipse.

- Open Eclipse, on the Menu bar, choose Window → Preferences → search User Libraries → New → Name it as "JavaFX"

- Click “Add External JARs” then navigate to the folder where you extracted JavaFX in the previous step, choose the “lib” folder and add all .jar files. Click “Apply and Close”.
- Right-click on the project → Build Path → Configure Build Path → Classpath → Add Library → User Library → JavaFX

1.4 Configuring Build Path and Arguments

You need to set up the run configuration by following these steps:

- Right-click on the project → Run As → Run Configurations → Arguments → VM arguments
- Add the following command: `--module-path "YOUR\PATH\lib" --add-modules javafx.controls,javafx.fxml`
  E.g: `--module-path "C:\javafx\openjfx-16_windows-x64_bin-sdk\javafx-sdk-16\lib" --add-modules javafx.controls,javafx.fxml`
- Click Apply.

2 Setup JavaFX Scene Builder for Eclipse

JavaFX Scene Builder is a visual layout tool that lets users quickly design JavaFX application user interfaces, without coding. Users can drag and drop UI components to a work area, modify their properties, apply style sheets, and the FXML code for the layout that they are creating is automatically generated in the background. The result is an FXML file that can then be combined with a Java project by binding the UI to the application’s logic.

2.1 The requires

- **e(fx)clipse** plugin is required to embed Scene Builder into Eclipse (already installed in part 1.1).

2.2 Download JavaFX Scene Builder

- Go to URL http://www.oracle.com/technetwork/java/javase/downloads/javafxscenebuilder-1x-archive-2199384.html
- Download the appropriate version to your computer and install

2.3 Configuring Eclipse to use the Scene Builder

- In eclipse select: Window/References

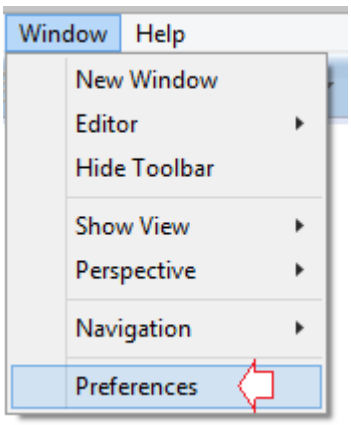


Figure 2. Configure Eclipse to use Scene Builder (1)

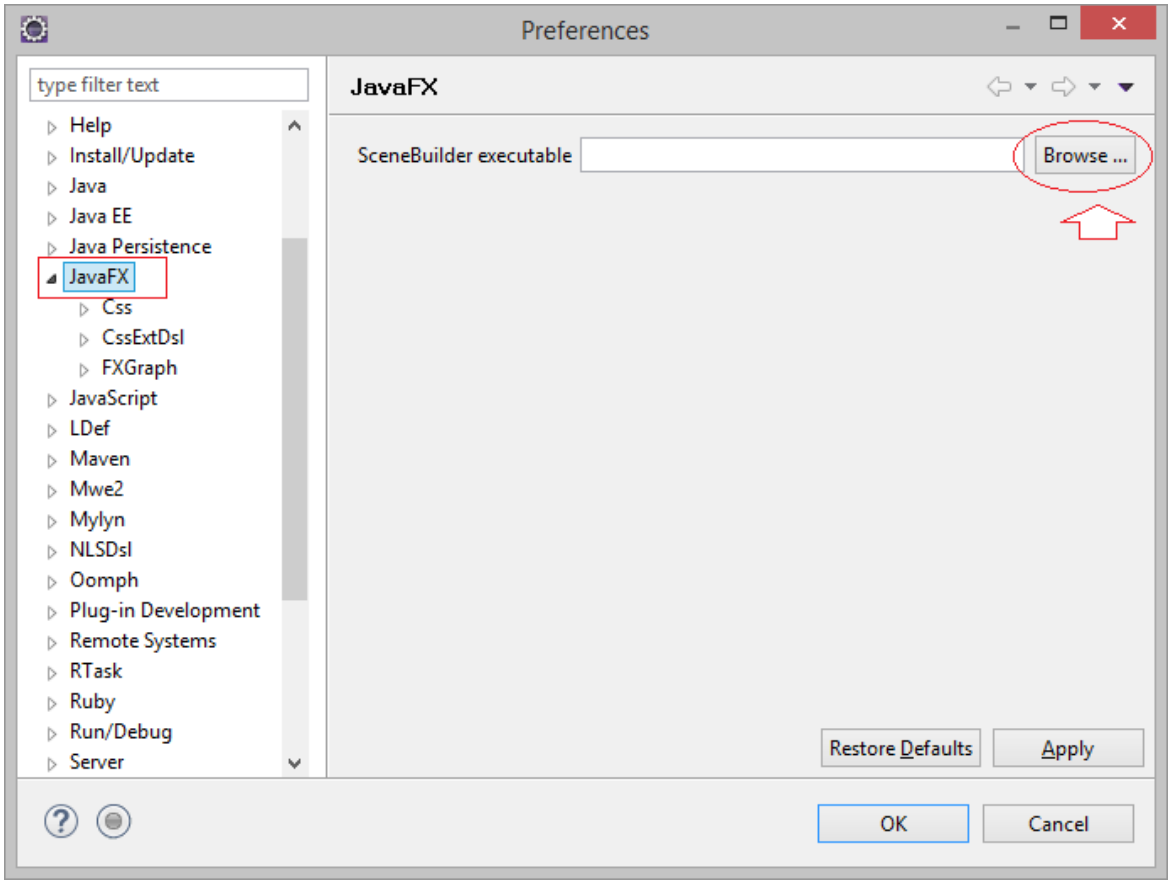


Figure 3. Configure Eclipse to use Scene Builder (2)

- Pointing to the exe file position of JavaFX Scene Builder.
-

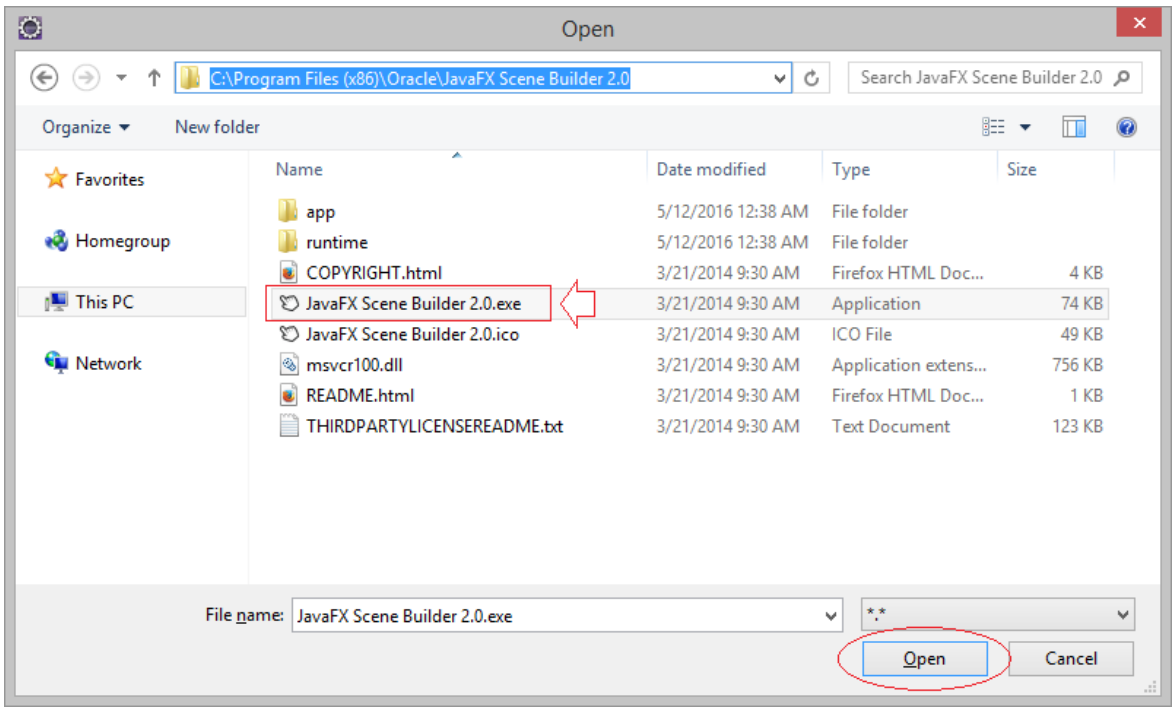


Figure 4. Configure Eclipse to use Scene Builder (3)

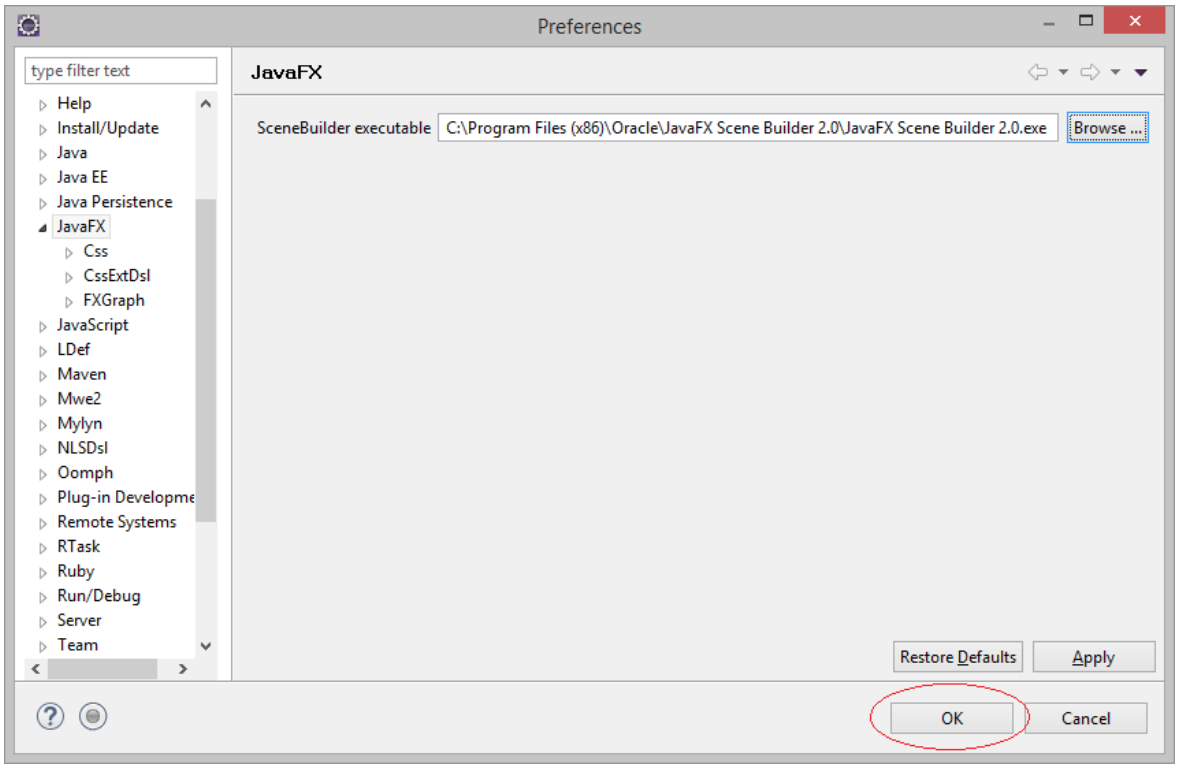


Figure 5. Configure Eclipse to use Scene Builder (4)

3 JavaFX API

Note: For the exercises in this lab (excluding the AIMS exercises), you will continue to use the `GUIProject`, and put all your source code in a package called "**hust.soict.globalict.javafx**" (for ICT) or "**hust.soict.dsai.javafx**" (for DS & AI). You might need to add the JavaFX library to this project if you are using the JDK version after 1.8.

In this exercise, we revisit the components of a JavaFX application by implementing a simple `Painter` app that allows the user to draw on a white canvas with their mouse.

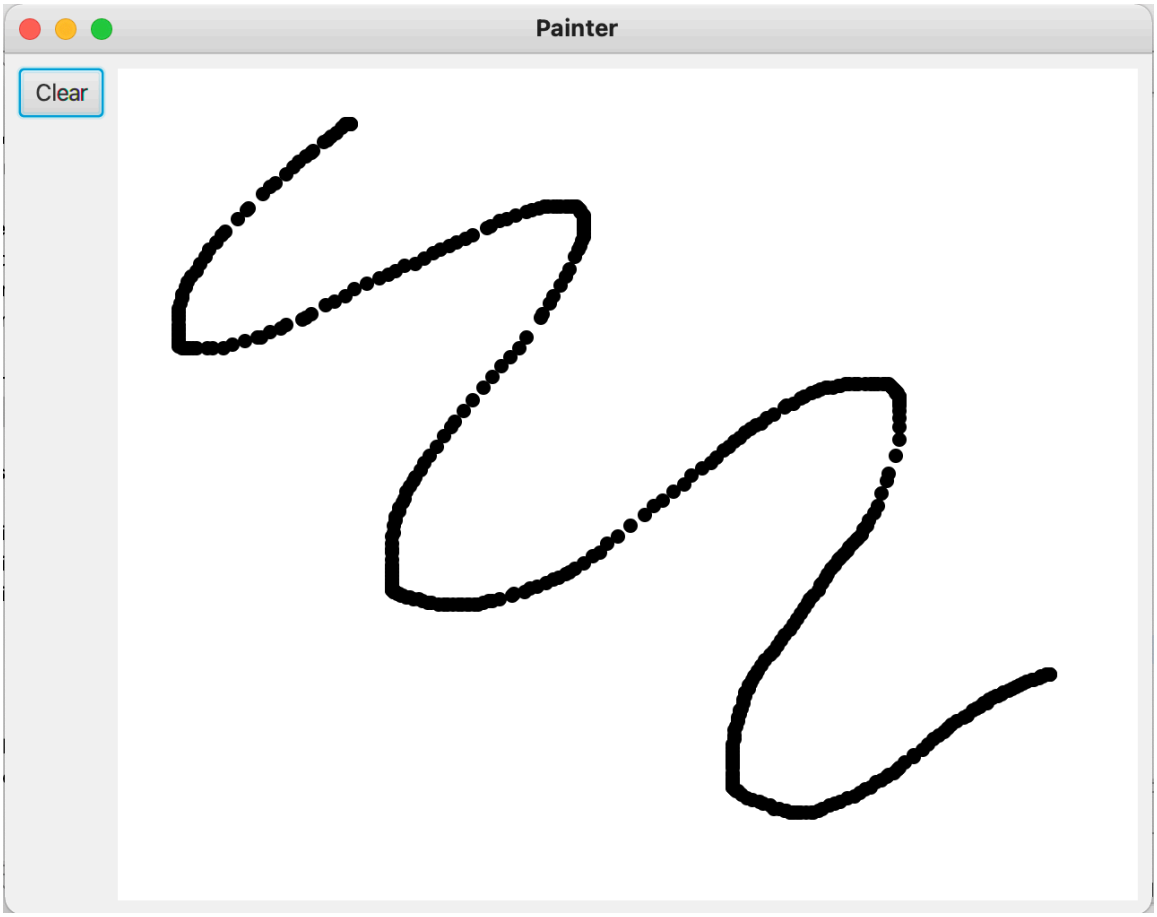


Figure 6. Painter app

Recall the basic structure of a JavaFX application: It uses the metaphor of a theater to model the graphics application. A stage (defined by the `javafx.stage.Stage` class) represents the top-level container (window). The individual controls (or components) are contained in a scene (defined by the `javafx.scene.Scene` class). An application can have more than one scene, but only one of the scenes can be displayed on the stage at any given time. The contents of a scene is represented in a hierarchical scene graph of nodes (defined by `javafx.scene.Node`).

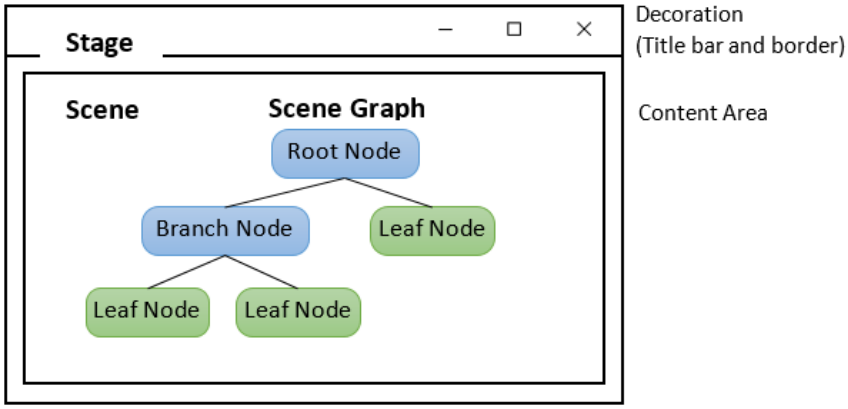


Figure 7. Structure of JavaFX application

Like any other JavaFX application, there are 3 steps for creating this `Painter` app as follows:

- Create the FXML file "`Painter.fxml`" (we will be using Scene Builder)
- Create the controller class `PainterController`
- Create the application class `Painter`

The FXML file lays out the UI components in the scene graph. The controller adds interactivity to these components by providing even-handling methods. Together, they complete the construction of the scene graph. Finally, the application class creates a scene with the scene graph and adds it to the stage.

3.1 Create the FXML file

3.1.1 Create and open the FXML file in Scene Builder from Eclipse:

Right-click on the appropriate package of `GUIProject` in Project Explorer. Select New > Other… > New FXML Document as in Figure 8.

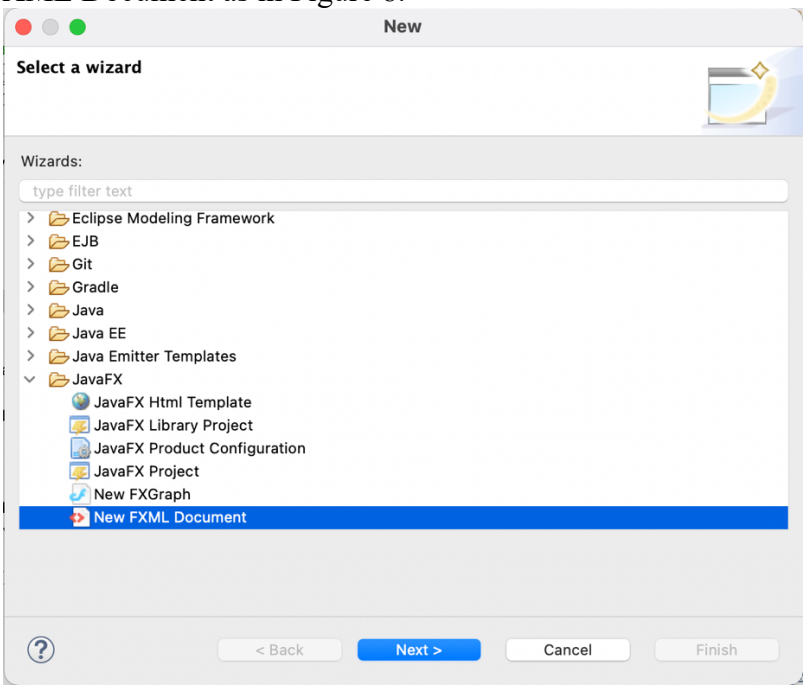


Figure 8. Create a new FXML in Eclipse(1)

Name the file "`Painter`" and choose `BorderPane` as the root element as in Figure 9

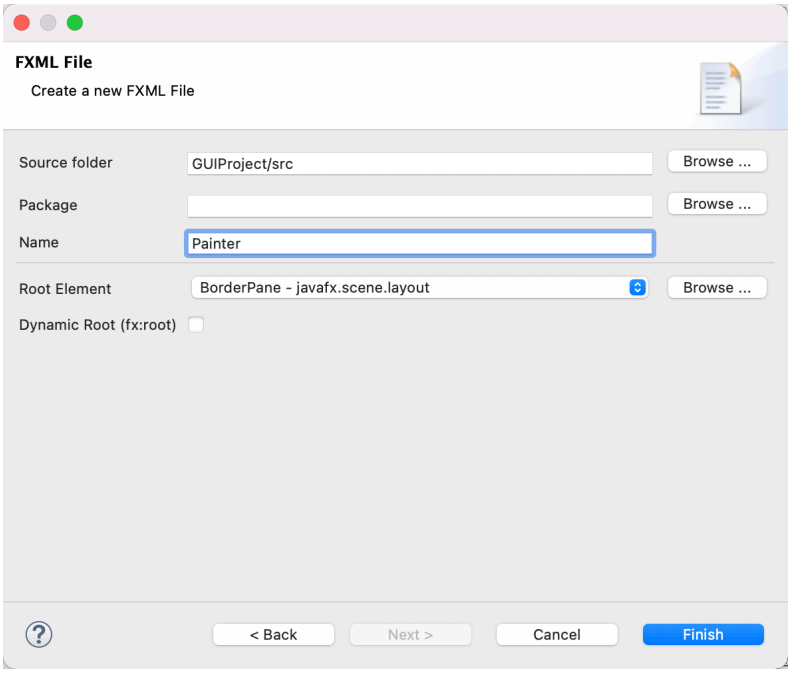


Figure 9. Create a new FXML in Eclipse(2)

A new file is created. Right-click on it in Project Explorer and select Open with SceneBuilder

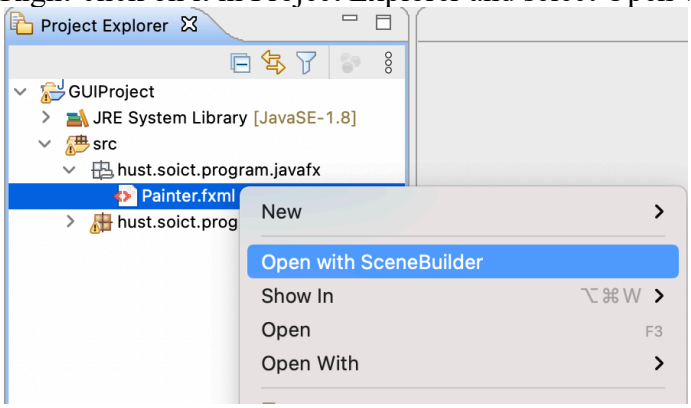


Figure 10. Open FXML with SceneBuilder from Eclipse

3.1.2 Building the GUI:

Our interface is divided into two sections: A larger section on the right for the user to paint on and a smaller section on the left which acts as a menu of tools and functionalities. For now, the menu only contains one button for the user to clear the board.

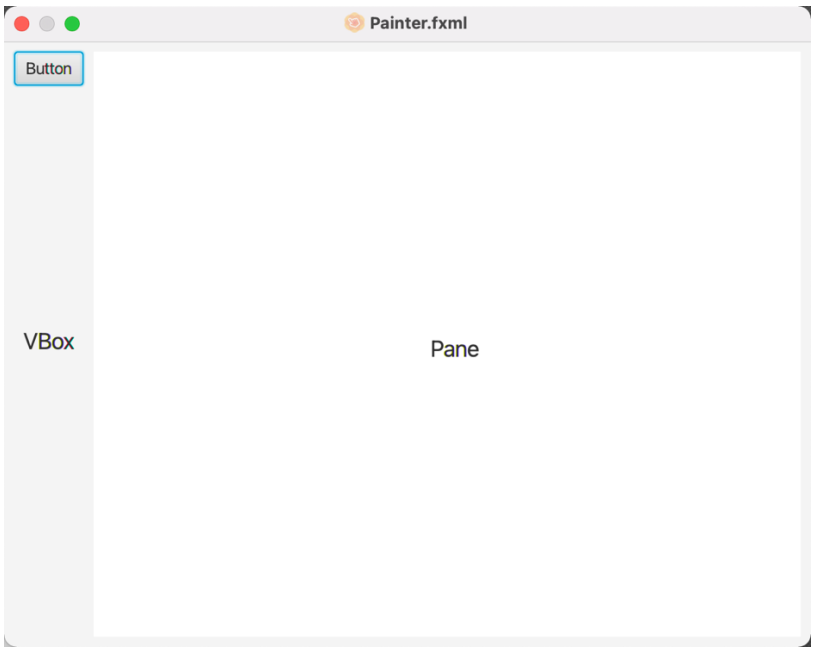


Figure 11. Target interface

For the right-side section, we use a regular `Pane`. On the other hand, for the left-side section, since we want to arrange subsequent items below the previous ones vertically, we use a `VBox` layout.

Step 1. Configuring the BorderPane – the root element of the scene

- We set the GridPane's Pref Width and Pref Height properties to 640 and 480 respectively. Recall that the stage's size is determined based on the size of the root node in the FXML document
- Set the BorderPane's Padding property to 8 to inset it from the stage's edges

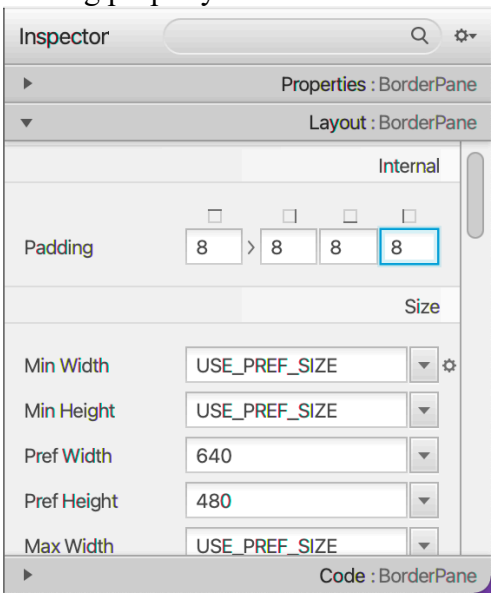


Figure 12. Configuring the BorderPane

Step 2. Adding the VBox

- Drag a VBox from the library on the left-hand side (you can search for VBox) into the BorderPane's LEFT area.

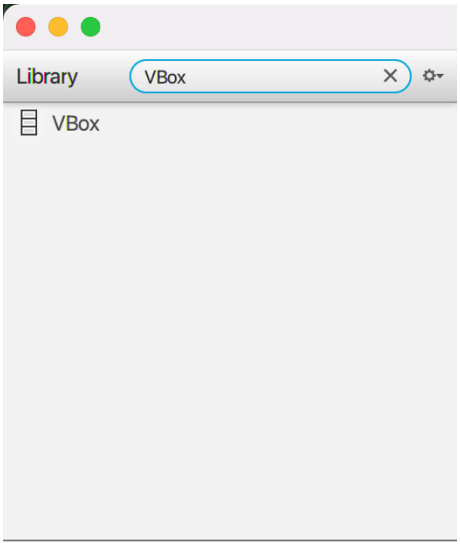


Figure 13. Get VBox in the Library menu

- Set the Pane's fx:id to **drawingAreaPane**.
- Set its Spacing property (in the Inspector's Layout section) to 8 to add some vertical spacing between the controls that will be added to this container (XXX)
- Set its right Margin property to 8 to add some horizontal spacing between the VBox and the Pane be added to this container (XXX)
- Also reset its Pref Width and Pref Height properties to their default values (USE_COMPUTED_SIZE) and set its Max Height property to MAX_VALUE. This will enable the VBox to be as wide as it needs to be to accommodate its child nodes and occupy the full column height (XXX)

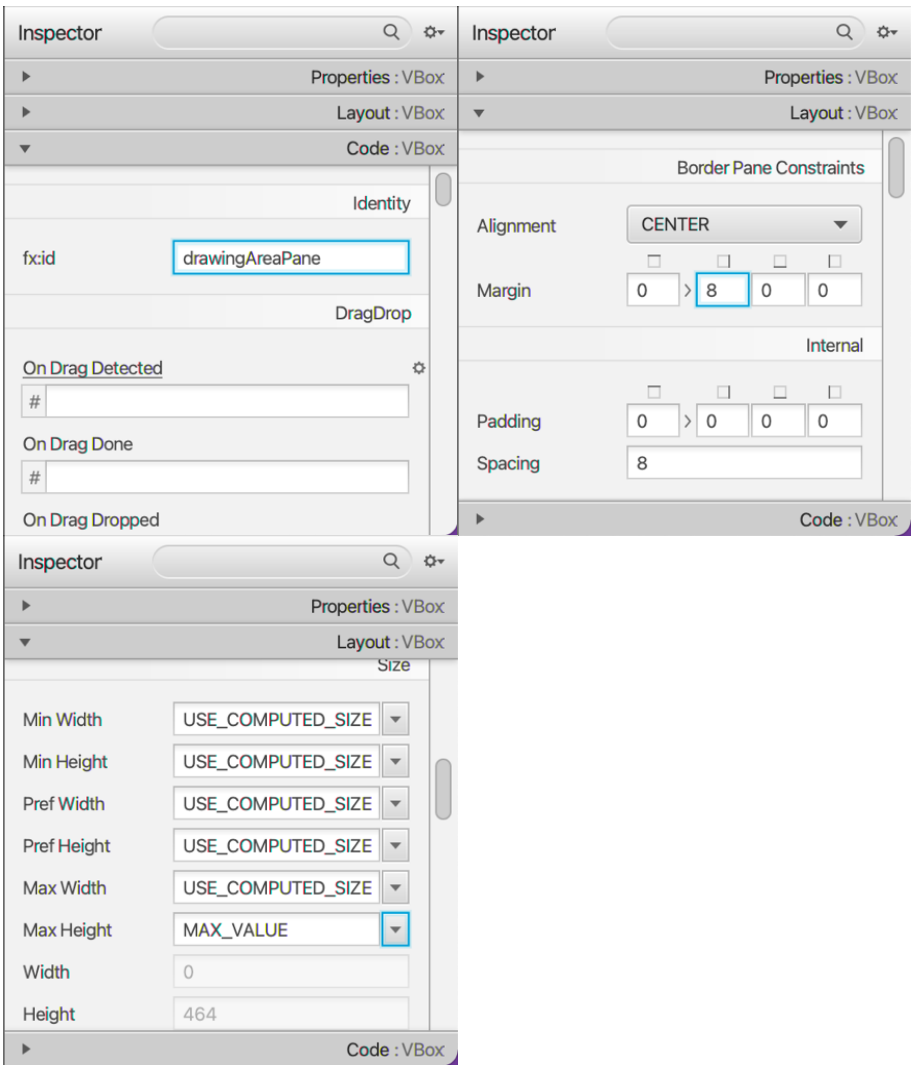


Figure 14. Configuring the VBox

Step 3. Adding the Pane

- Drag a Pane from the library on the left hand-side into the BorderPane's CENTER area.
- In the JavaFX CSS category of the Inspector window's Properties section, click the field below Style (which is initially empty) and select -fx-background-color to indicate that you'd like to specify the Pane's background color. In the field to the right, specify white.
- Specify drawingAreaMouseDragged as the On Mouse Dragged event handler (located under the Mouse heading in the Code section). This method will be implemented in the controller.

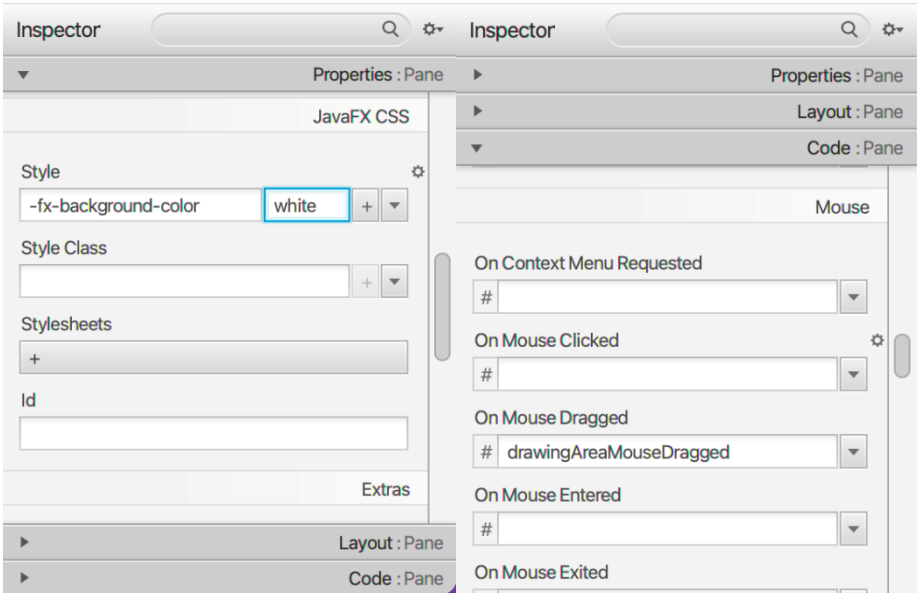


Figure 15. Configuring the Pane

Step 4. Adding the Button

- Drag a Button from the library on the left hand-side into the VBox.
- Change its text to "Clear" and set its Max Width property to MAX_VALUE so that it fills the VBox's width.
- Specify clearButtonPressed as the On Action event handler. This method will be implemented in the controller.

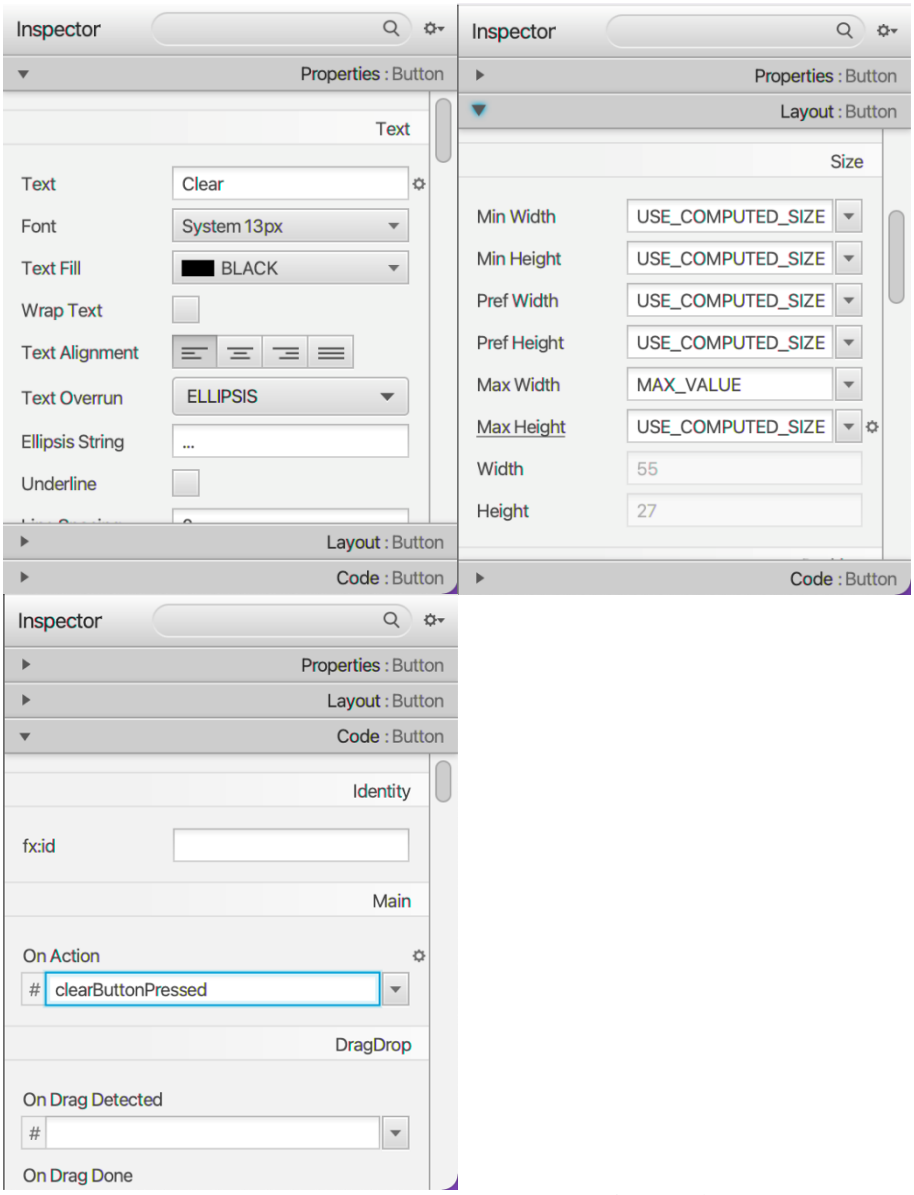


Figure 16. Configuring the Button

Now that all the elements are set, you can preview them by selecting Preview > Show Preview in Window

3.2 Create the controller class

In the same package as the FXML, create a Java class called `PainterController`. You can also utilize Scene Builder for coding the controller as follows: Select View > Show Sample Controller Skeleton. A window like in XXX will appear:

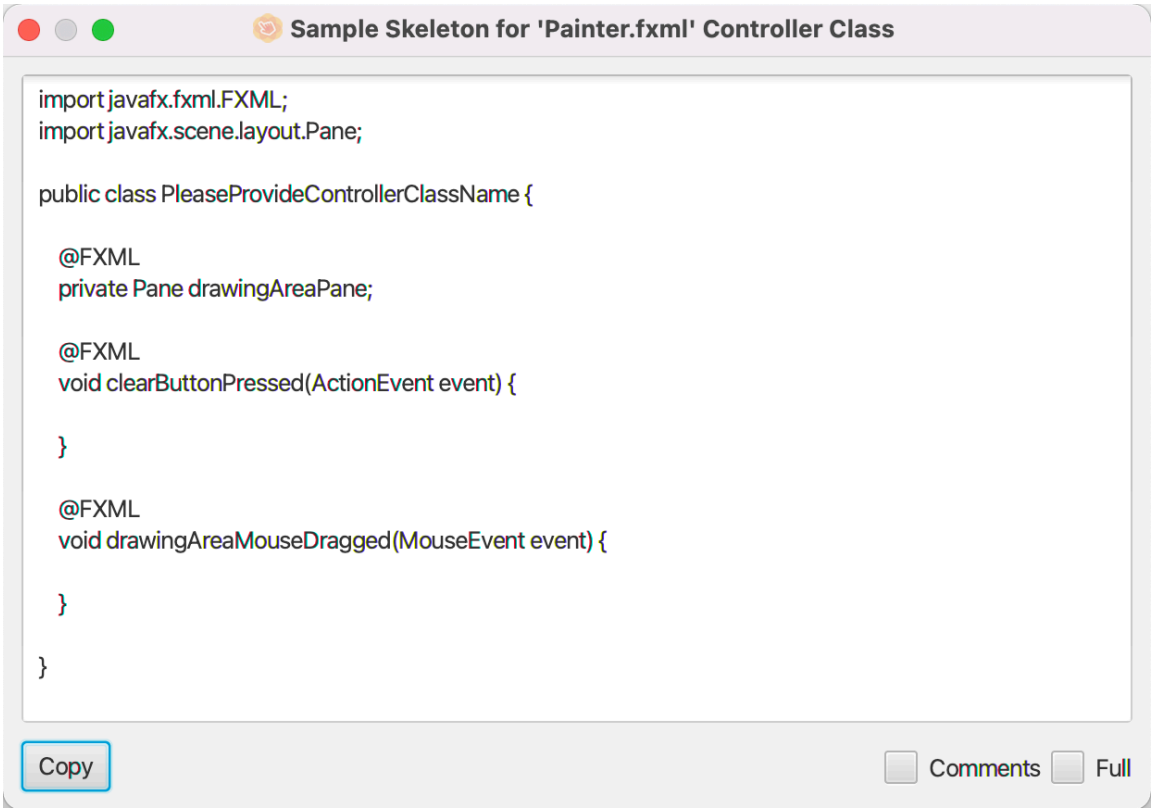


Figure 17. Auto-generated skeleton code for the controller

You can choose to copy the skeleton and paste it into your `PainterController.java` file. Remember to replace the class name in the skeleton code with your actual class name (`PainterController`). The results look roughly like this:

```
 8  public class PainterController {
 9
10      @FXML
11      private Pane drawingAreaPane;
12
13      @FXML
14      void clearButtonPressed(ActionEvent event) {
15          //implement clearing of canvas here
16      }
17
18      @FXML
19      void drawingAreaMouseDragged(MouseEvent event) {
20          //implement drawing here
21      }
22
23  }
```

Figure 18. Skeleton copied into PainterController

Next, we will implement the event-handling functions.

For the `drawingAreaMouseDragged()` method, we determine the coordinate of the mouse through `event.getX()` and `event.getY()`. Then, we add a small `Circle` (approximating

a dot) to the `Pane` at that same position. We do this by getting the `Pane`'s children list – which is an `ObservableList` - and adding the UI object to the list.

```
20      @FXML
21      void drawingAreaMouseDragged(MouseEvent event) {
22          Circle newCircle = new Circle(event.getX(),
23                  event.getY() , 4, Color.BLACK);
24          drawingAreaPane.getChildren().add(newCircle);
25      }
```

Figure 19.Source code of drawingAreaMouseDragged()

For the `clearButtonPressed()` method, we simply need to clear all the `Circle` objects on the `Pane`. Again, we have to access the `Pane`'s children list through `Pane.getChildren()`.

```
15      @FXML
16      void clearButtonPressed(ActionEvent event) {
17          drawingAreaPane.getChildren().clear();
18      }
```

Figure 20. Source code of clearButtonPressed()

The source code for the controller is complete, however, to ensure that an object of the controller class is created when the app loads the FXML file at runtime, you must specify the controller class's name in the FXML file using Scene Builder, in the lower right corner under Document menu > Controller.

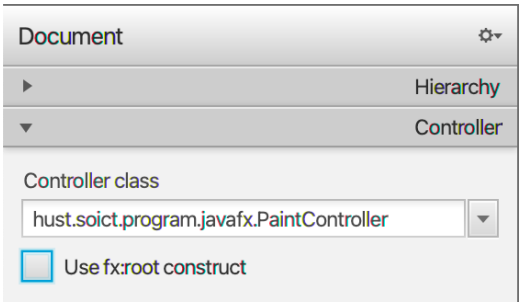


Figure 21. Specify the controller for the FXML file in Scene Builder

3.3 Create the application

Create a class named `Painter` in the same package as the FXML and the controller class. The source code is provided below:

```
 9  public class Painter extends Application{
10
11      @Override
12      public void start(Stage stage) throws Exception {
13          Parent root = FXMLLoader.load(getClass()
14                  .getResource("/hust/soict/program/javafx/Painter.fxml"));
15
16          Scene scene = new Scene(root);
17          stage.setTitle("Painter");
18          stage.setScene(scene);
19          stage.show();
20
21      }
22      public static void main(String[] args) {
23          launch(args);
24      }
25  }
```

Figure 22. Painter source code

Explanation of the code:

- All JavaFX applications must extend the `Application` class.
- `main()` method:
  In the `main` method, the `launch` method is called to launch the application. Whenever an application is launched, the JavaFX runtime does the following, in order:
  - Constructs an instance of the specified Application class
  - Calls the `init` method
  - Calls the `start` method
  - Waits for the application to finish
  - Calls the `stop` method

  Note that the `start` method is abstract and must be overridden. The `init` and `stop` methods have concrete implementations that do nothing.
- `start()` method:
  Here, in the `start` method a simple window is set up by loading the FXML into the root node. From that root node, a `Scene` is created and set on the `Stage`.

3.4 Practice exercise:

3.4.1 Draw when mouse down

In the current version of the `Painter` app, if the user just presses down on the mouse without dragging it, nothing will appear on the canvas (because we only add the handling method for the "On Mouse Dragged" entry for the Pane). The expected output should be a dot appearing at the position of the mouse.

Your task is to improve the Painter application so the output will be as the one expected. You shouldn't have to change your source code for this, rather just the FXML file through the Scene Builder GUI. Since the event-handling for mouse pressed and mouse dragged are the same, they can share the same event-handler method (drawingAreaMouseDragged). You just need to pick the appropriate entry among the ones below to set the event-handler method name to. (On Mouse Dragged and another entry will have the same method name).

Note: After modifying the FXML file, you might need to refresh your Java project to make sure the latest changes are updated. You can do so by right-clicking on your project in Project Explorer > Refresh.

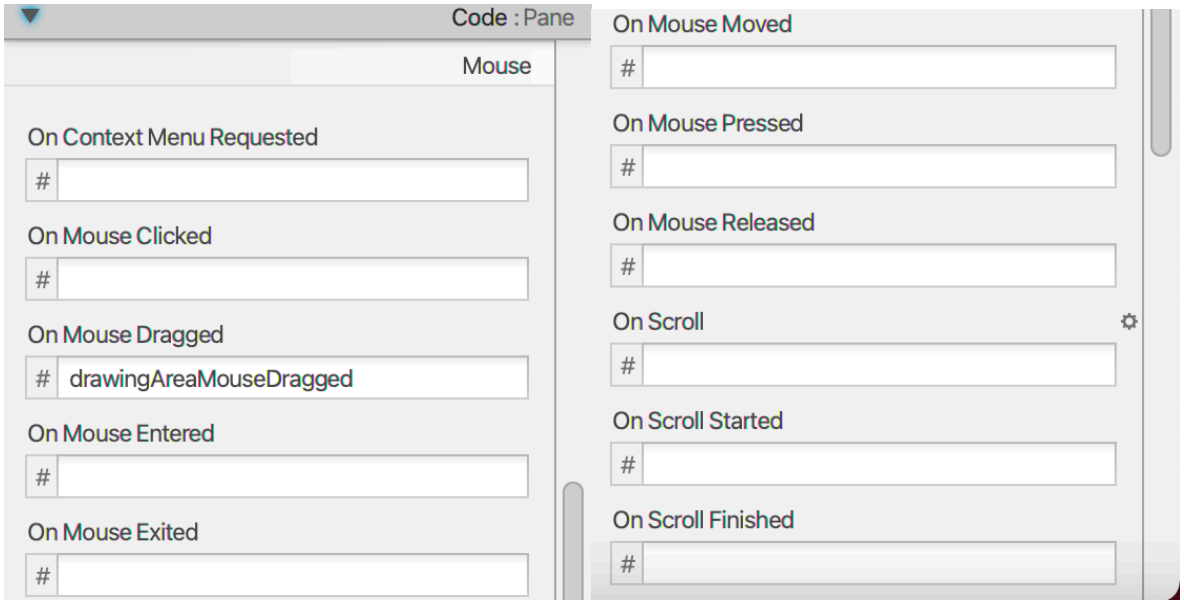


Figure 23. Choose another MouseEvent handling entry

3.4.2 Add the Eraser functionality

The new interface of the app including the new eraser functionality should be like this:

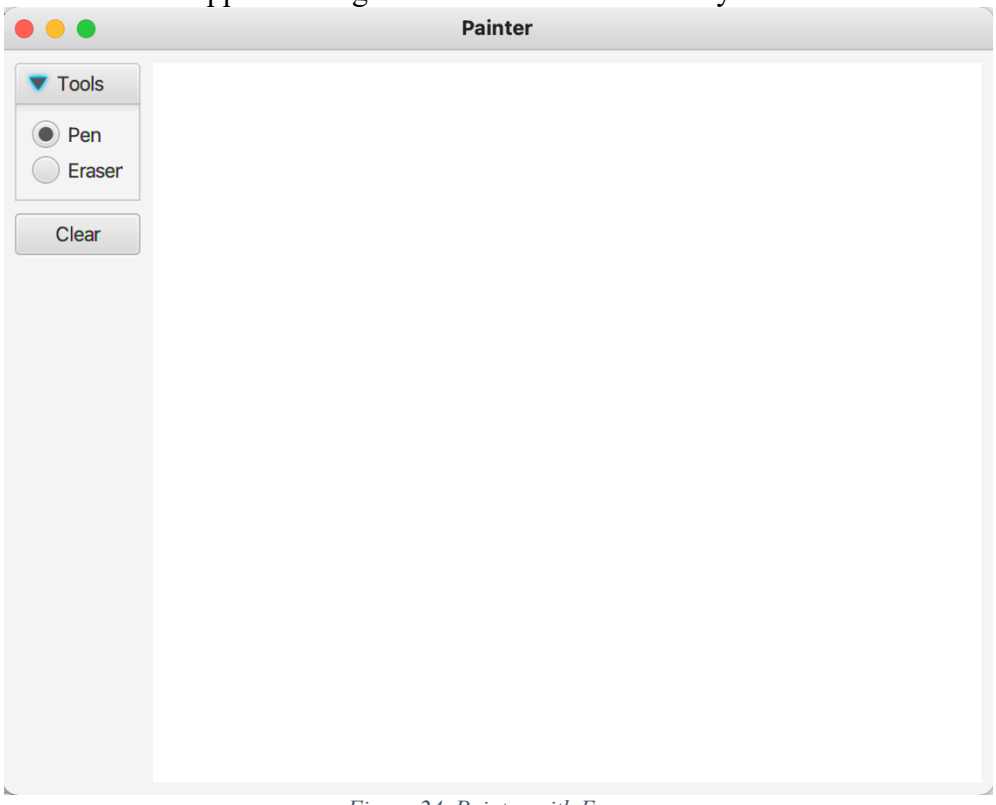


Figure 24. Painter with Eraser

Hint:

- For the interface design: use `TitledPane` and `RadioButton`. Using Scene Builder, set the Toggle Group properties of the RadioButtons as identical, so only one of them can be selected at a time.

- For the implementation of Eraser: One approach is to implement an eraser just like the pen above, but use white ink color (canvas color) instead.

4 Requiremts of AIMS Customer Application

In the previous lab, we already created the AIMS application for store manager by using Swing. In this lab, we will build AIMS application for customer with JavaFX.
With this application, customer can perform the following functions:

- View store: Customer can view all media in the store, play a media if this media is DVD or CD and add a media to the cart
- For cart: Customer can view the cart, see the total cost, filter (search) media, play a DVD/CD, remove a media from the cart, and place order.

Note: Under the `src` folder, create 3 new packages as follow

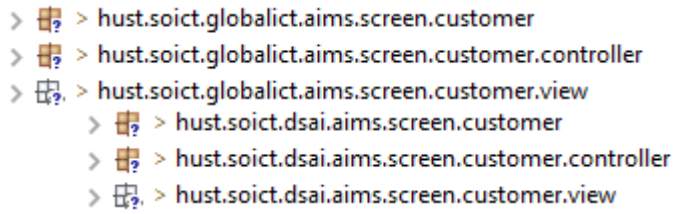


Figure 25. New packages for ICT class

Figure 26. New packages for DSAI class

- Package `customer` stores the main class of application for customer
- Subpackage `view` stores FXML files and subpackage `controller` stores the controllers of those FXML files.

5 Build View Store Screen for AIMS Customer Application with JavaFX

5.1 Set up View Store Screen with Scene Builder

Figure 27. View store screen for Customer

Like the previous exercise, we start by creating an FXML file named "**Store.fxml**" in the package `screen.customer.view` with VBox being the root node. The View Store Screen contains two main parts:

- An **Hbox** contains a label AIMS and a button View Cart
- A **ScrollPane** contains GridPane to show all the items in the store

The structure of Store.fxml file is as shown below:

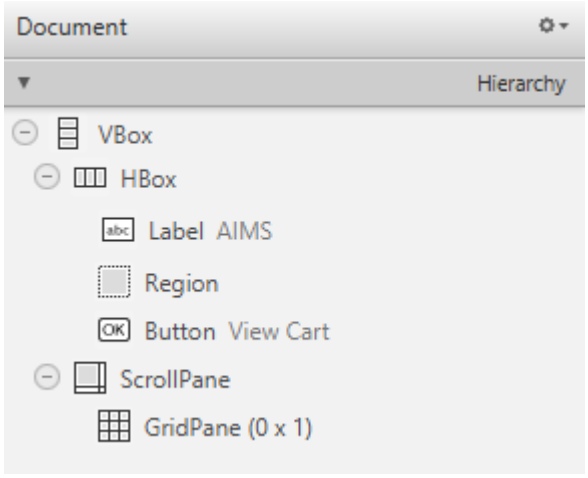


Figure 28. Structure of Store.fxml file

In this section, we will create the GUI components (i.e, the FXML file) only. The controller of ViewStoreScreen will be implemented later.

5.1.1 Set up the VBox

- Properties:
  - Alignment: CENTER
- Layout:
  - Pref Width: 1024
  - Pref Height: 768
  - Padding: 20 Left

5.1.2 Set up the HBox

Step 1. Set up HBox Properties and Layout

- Properties:
  - Alignment: CENTER
- Layout:
  - Pref Height: 100

Step 2. Add a Label to the HBox

- Properties:
  - Text: AIMS
  - Font: 50px
  - Text Fill: #004cff
- Layout:
  - Padding: 10 Left

Step 3. Add a Region to the HBox

- Layout:
  - Hgrow: ALWAYS

Step 3. Add a Button to the HBox

- Properties:
  - Text: View Cart
- Layout:
  - Margin: 20 Right
  - Pref Width: 100
  - Pref Height: 50

- Code:
    - On Action: btnViewCartPressed

5.1.3 Set up the ScrollPane:

Step 1. Add a ScrollPane to the VBox, under the HBox

- Layout:
    - Pref Width: 1024
    - Pref Height: 760

Step 2. Add a GridPane to the ScrollPane

- Because the number of items in the store can be changed, therefore we will create a dynamic GridPane that updates the view along with the change of the Store.
- Firstly, we will create an "empty" GridPane. Drag a GridPane to the ScrollPane. Delete given rows and columns in the GridPane by right click on the row/column to be deleted, choose Delete. The final dimension of the GridPane is (0 x 1).

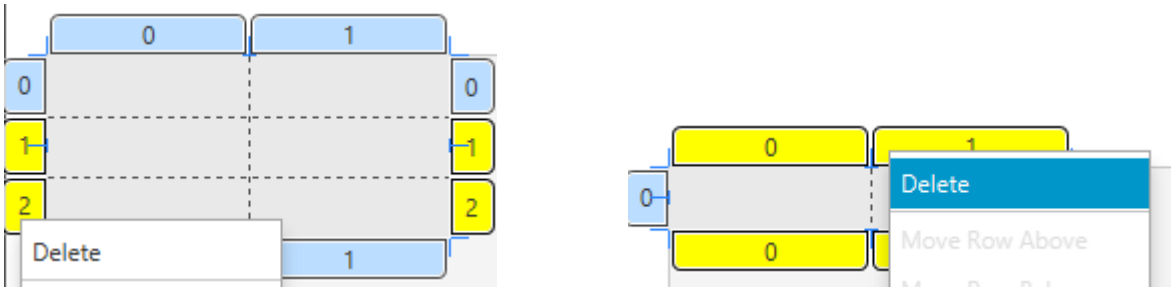


Figure 29. Configure GridPane

- Set up Layout:
    - Pref Width: 0
    - Pref Height: 0
- Code:
    - fx:id: gridPane

5.2 Set up Item in the Store

In the View Store Screen, we already created an empty GridPane. In this section, we will create an FXML component to display the information of media and dynamically add it to the GridPane.

5.2.1 Create Item.fxml file

In package `screen.customer.view`, create the Item.fxml file, choose the root node to be AnchorPane.

The structure of this file is shown as below:

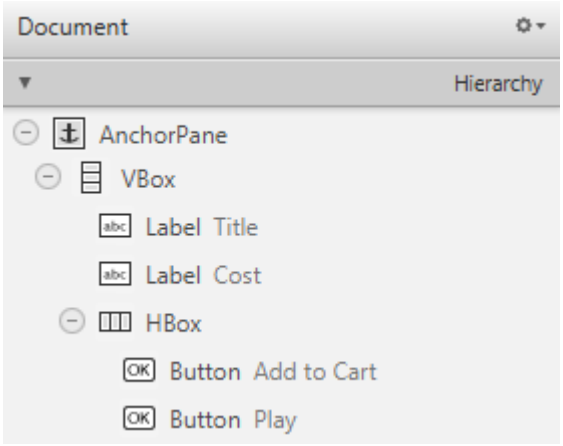
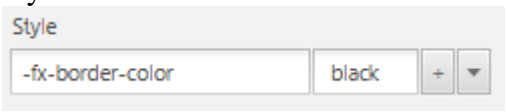


Figure 30. Structure of Item.fxml file

Step 1. Set up the AnchorPane

- Properties:
  - Style:



Step 2. Add a Vbox to the AnchorPane

- Properties:
  - Alignment: CENTER
- Layout:
  - Pref Width: 305
  - Pref Height: 175

Step 3. Add two Labels to the VBox

- Properties:
  - Alignment: CENTER
- Layout:
  - Pref Width: 305
  - Pref Height: 50
- Code:
  - Label 1: set fx:id to lblTitle
  - Label 2: set fx:id to lblCost

Step 4. Add a HBox to the Vbox

- Properties:
  - Alignment: CENTER
- Layout:
  - Pref Width: 305
  - Pref Height: 50
  - Spacing: 20

Step 5. Add two Buttons to the HBox

❖ Button 1:

- Properties:
  - Text: Add to Cart
- Layout:

        - Pref Width: 305
        - Pref Height: 50
- Code:
    - fx:id: btnAddToCart
    - On Action: btnAddToCartClicked

❖ Button 2:

- Properties:
    - Text: Play
- Layout:
    - Pref Width: 305
    - Pref Height: 50
- Code:
    - fx:id: btnPlay
    - On Action: btnPlayClicked

5.2.2 Create ItemController class

The fxml file just contains the GUI, not the logic. You need to create a controller to implements the behaviors of the FXML components.

In the package `customer.screen.controller`, create the ItemController class. You can copy the Controller Skeleton from Scene Builder by select View > Show Sample Controller Skeleton.

Add the following attribute and method to the ItemController class:

```
44      private Media media;
45      public void setData(Media media) {
46          this.media = media;
47          lblTitle.setText(media.getTitle());
48          lblCost.setText(media.getCost()+" $");
49          if(media instanceof Playable) {
50              btnPlay.setVisible(true);
51          }
52          else {
53              btnPlay.setVisible(false);
54              HBox.setMargin(btnAddToCart, new Insets(0, 0, 0, 60));
55          }
56      }
```

Figure 31. setData() method in class ItemController

Explain the code:

- Lines 46-48 are used to set the information of the media
- Lines 49-55 updates the visibility of the Play button depending on the type of media

5.3 Create ViewStoreController class

In the package `customer.screen.controller`, create the ViewStoreController class. Add all attributes and methods defined in the *Store.fxml* file to the Controller. You can copy the Controller Skeleton from Scene Builder by select View > Show Sample Controller Skeleton.

Declare one attribute in the `StoreManagerScreen` class: `Store store` (because we need information of the items in the store to display them) and and pass it to the constructor.

```
public class ViewStoreController {
    private Store store;
    public ViewStoreController(Store store) {
        this.store = store;
    }
```

Figure 32. Declaration of ViewStoreController class

The `GridPane` we created in the section 5.1 is currently empty, we need to fill it with data of media items in the store. In the `initialize()` method, we load the fxml file of each media item to an FXML component (lines 35-40), set data for the item component (line 41) and add it to the GridPane (line 48).

```
28      @FXML
29      public void initialize() {
30          final String ITEM_FXML_FILE_PATH = "/hust/soict/program/aims/screen/customer/view/Item.fxml";
31          int column = 0;
32          int row = 1;
33          for(int i=0; i<store.getItemsInStore().size(); i++) {
34              try {
35                  FXMLLoader fxmlLoader = new FXMLLoader();
36                  fxmlLoader.setLocation(getClass().getResource(ITEM_FXML_FILE_PATH));
37                  ItemController itemController = new ItemController();
38                  fxmlLoader.setController(itemController);
39                  AnchorPane anchorPane = new AnchorPane();
40                  anchorPane = fxmlLoader.load();
41                  itemController.setData(store.getItemsInStore().get(i));
42
43                  if (column == 3) {
44                      column = 0;
45                      row++;
46                  }
47
48                  gridPane.add(anchorPane, column++, row);
49                  GridPane.setMargin(anchorPane, new Insets(20, 10, 10, 10));
50              } catch (IOException e) {
51                  e.printStackTrace();
52              }
53          }
54      }
```

Figure 33. Source code for initialize() method of ViewStoreController

Explain initialize() method: The FXML controller can define an `initialize()` method, which will be called once on an implementing controller when the contents of its associated

document have been completely loaded. In a few words: The constructor is called first, then any @FXML annotated fields are populated, then `initialize()` is called.

5.4 Test View Store Screen

To test if our previouse code actually works, we will create a new package under the src pakcage named: `hust.soict.program.test.screen.customer.store`, where `program` is replaced by `globalict` or `dsai`.

In that package, create a class `TestViewStoreScreen` that extends `Application` class with the main method. The sample code is shown below:

```
11 public class TestViewStoreScreen extends Application {
12     private static Store store;
13 
14     @Override
15     public void start(Stage primaryStage) throws Exception {
16         final String STORE_FXML_FILE_PATH = "/hust/soict/program/aims/screen/customer/view/Store.fxml";
17         FXMLLoader fxmlLoader = new FXMLLoader(getClass().getResource(STORE_FXML_FILE_PATH));
18         ViewStoreController viewStoreController = new ViewStoreController(store);
19         fxmlLoader.setController(viewStoreController);
20         Parent root = fxmlLoader.load();
21 
22         primaryStage.setTitle("Store");
23         primaryStage.setScene(new Scene(root));
24         primaryStage.show();
25     }
26 
27     public static void main(String[] args) {
28         store = new Store();
29         /*
30          * Add some items to store here
31          * ...
32         */
33         launch(args);
34     }
```

Figure 34. Sample code to test View Store Screen

6 Build Cart for AIMS Customer Application

6.1 Set up Cart Screen with Scene Builder

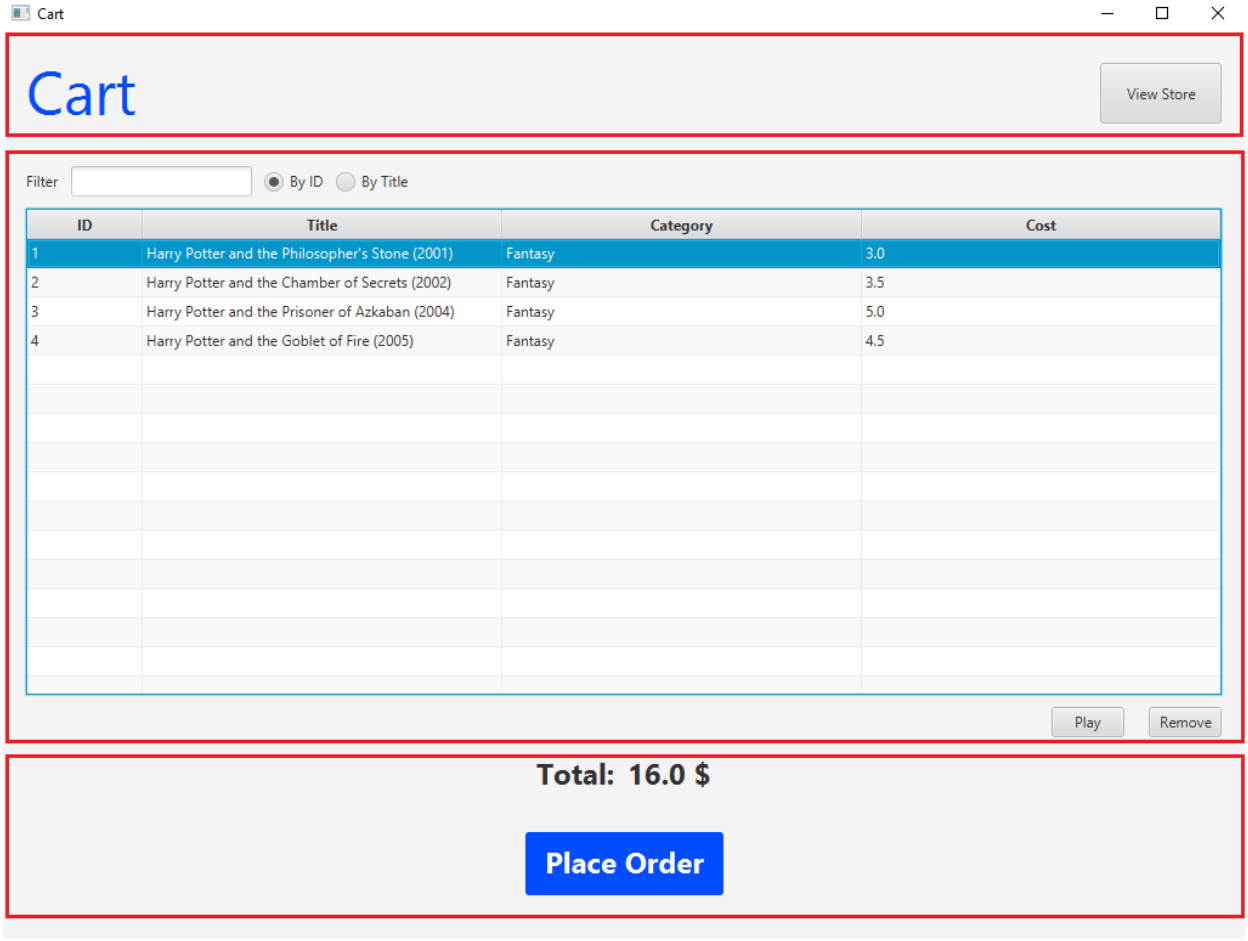


Figure 35. View cart screen for Customer

Similar to the View Store Screen, in package `screen.customer.view`, we will first create a `Cart.fxml` file with BorderPane being the root node. The Cart Screen has three distinct areas (bounded in red borders) corresponding to TOP, CENTER and BOTTOM areas of BorderPane.

6.1.1 Set up the BorderPane:

- Layout:
  - Pref Width: 1024
  - Pref Height: 768

6.1.2 Set up the TOP area

We use the HBox layout for the TOP area to arrange components horizontally. The structure of TOP area is shown as below:

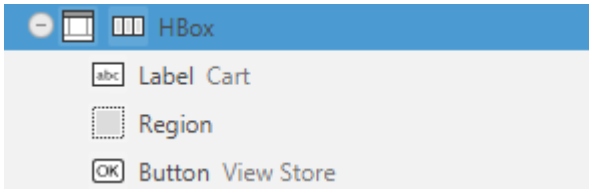


Figure 36. Structure of TOP area

Step 1. Drag a HBox into the BorderPane's TOP area.
- Layout:
  - Margin: 20 Left, 20 Right
  - Pref Height: 100

Step 2. Add a Label to the HBox
- Properties:
  - Text: CART
  - Font: 50px
  - Text Fill: #004cff

Step 3. Add a Region to the HBox
- Layout:
  - Hgorw: ALWAYS

Step 4. Add a Button to the HBox
- Properties:
  - Pref Width: 100
  - Pref Height: 50
- Code:
  - On Action: btnViewStorePressed

6.1.3 Setg up the CENTER area

For the CENTER area, we use the VBox layout to arrange components vertically. Inside the Vbox, we use an HBox to arrange the top row of components horizontally. We also use a TableView to display data in a tabular form. Below the TableView is another HBox containing two buttons Play and Remove.

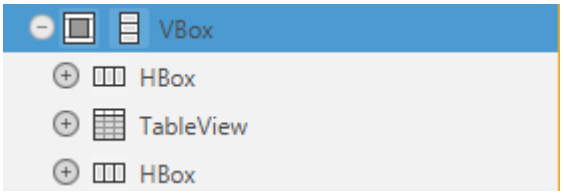


Figure 37. Structure of CENTER area

Step 1. Drag a VBox into the CENTER area
- Layout:
  - Alignment: CENTEr
  - Margin: 20 Left, 20 Right

Step 2. Add an HBox into the Vbox

Step 2.1. Set up the HBox
- Properties:
  - Alignment: CENTER_LEFT
- Layout:
  - Padding: 10 top & bottom
  - Spacing: 10

Step 2.2. Add a Label to the HBox
- Properties:
  - Text: Filter

Step 2.3. Add a TextField into the HBox

Step 2.4. Add the first RadioButtons into the HBox
- Properties:

        - Text: By ID
        - Selected: ✓
        - Toggle Group: filterCategory

    Step 2.5. Add the second RadioButtons into the HBox
    - Properties:
        - Text: By Title
        - Toggle Group: filterCategory

Step 3. Add a TableView into the VBox
- Set TableView's fx:id property to **tblMedia**
- Add for TableColumns
    - The Title column:
        - Text: ID
        - fx:id: **colMediaId**
    - The Title column:
        - Text: Title
        - fx:id: **colMediaTitle**
    - The Category column:
        - Text: Category
        - fx:id: **colMediaCategory**
    - The Cost column:
        - Text: Cost
        - fx:id: **colMediaCost**

Step 4. Add another HBox below the TableView into the Vbox

Step 4.1. Set up the HBox
- Properties:
    - Alignment: TOP_RIGHT
- Layout:
    - Padding: 10 Top
    - Spacing: 20

Step 4.2. Add button Play into the HBox
- Properties:
    - Text: Play
- Layout:
    - Pref Width: 60
- Code:
    - fx:id: btnPlay
    - On Action: btnPlayPressed

Step 4.3. Add button Remove into the HBox
- Properties:
    - Text: Remove
- Layout:
    - Pref Width: 60
- Code:
    - fx:id: btnRemove
    - On Action: btnRemovePressed

6.1.4 Setting up the BOTTOm area

Figure 38. Structure of CENTER area

Step 1. Drag a VBox into the BOTTOM area
- Properties:
  - Alignment: TOP_CENTER
- Layout:
  - Spacing: 30
  - Pref Height: 200

Step 2. Add an HBox into the VBox
- Properties: Alignment: CENTER
- Layout: Pref Width: USE_COMPUTED_SIZE
- Layout: Pref Height: USE_COMPUTED_SIZE

Step 3. Add a Label to the HBox
- Properties:
  - Text: Total:
  - Font: 24px Bold
- Layout:
  - Spacing: 10

Step 4. Add another Label to the HBox
- Properties:
  - Text: 0 $
  - Font: 24px Bold
- Code:
  - fx:id: costLabel

Step 5. Add a Button to the VBox
- Properties:
  - Text: Place Order
  - Font: 24px Bold
  - Text Fill: #ffffff (WHITE)
  - Style: -fx-background-color: #004cff

6.2 Create CartController class

In the package `customer.screen.controller`, create the CartController class. Add all attributes and methods defined in the *Cart.fxml* file to the Controller. You can copy the Controller Skeleton from Scene Builder by select View > Show Sample Controller Skeleton.

Declare one attribute in the `CartController` class: `Cart cart` (because we need information of the items in the cart to display them) and and pass it to the constructor.

```
public class CartController {

    private Cart cart;

    public CartController(Cart cart) {
        this.cart = cart;
    }
```

Figure 39. Declaration of CartController class

6.3 View the items in the cart – JavaFX's data-driven UI

The `TableView` we created in the earlier is currently empty, we need to fill it with data of media items in our cart. Similar to the View Store Controller, we will fill the data in `initialize()` method.

```
67      @FXML
68      public void initialize() {
69          colMediaId.setCellValueFactory(
70                  new PropertyValueFactory<Media, Integer>("id"));
71          colMediaTitle.setCellValueFactory(
72                  new PropertyValueFactory<Media, String>("title"));
73          colMediaCategory.setCellValueFactory(
74                  new PropertyValueFactory<Media, String>("category"));
75          colMediaCost.setCellValueFactory(
76                  new PropertyValueFactory<Media, Float>("cost"));
77          if(cart.getItemsOrdered() != null )
78              tblMedia.setItems(cart.getItemsOrdered());
```

Figure 40. initialize() method in CartController

In line 78, we set the cart's list of items to the items of the `TableView`. Note that this will initially cause an error, because we cannot set a regular `List` as the items of a `TableView`. Instead, we have to use an `ObservableList`, so that any change in the data can be observed and reflected by the `TableView`. Please open the source code of `Cart` and change the `itemsOrdered` from `List<Media>` to `ObservableList<Media>`

```
private ObservableList<Media> itemsOrdered =
        FXCollections.observableArrayList();
```

Figure 41. New itemsOrdered

After setting the items of the `TableView`, the data still isn't showing up in the `TableView` yet, because we still have to set up the way the columns can retrieve data. This is done by setting the columns' `cellValueFactory`.

In lines 69 – 76, we set the columns' `cellValueFactory` using the class `PropertyValueFactory<S, T>` (Read the Javadocs for more details). This class is a callback that will take in a `String <property>` and look for the method

`get<property>()` in the `Source S` class. If a method matching this pattern exists, the value returned from this method is returned to the `TableCell`.
You can now test the Cart Screen with some media in your cart, the results will look roughly like this:

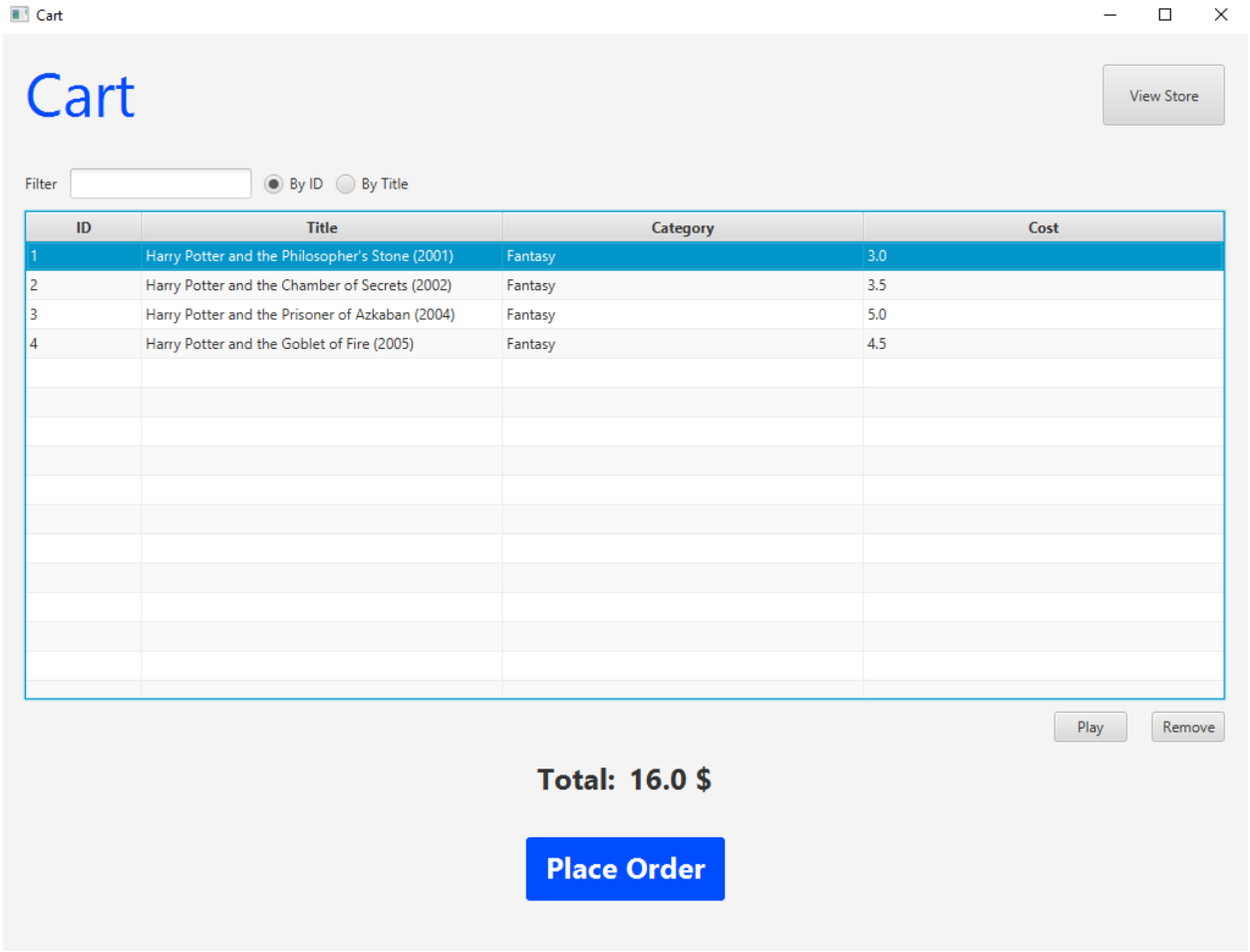


Figure 42. Test displaying TableView data

6.4 Updating buttons based on the selected item in `TableView` – `ChangeListener`

We will now implement a button bar (containing Play and Remove button) that changes buttons based on the `Media` currently selected in the `TableView`. Make sure that you add the fx:id property for two buttons in the fxml file (you can check in Scene Builder) and create the corresponding attributes in `CartController`:

- The "Play" Button: **btnPlay**
- The "Remove" Button: **btnRemove**

First of all, the buttons only appear when a certain `Media` object is being selected, which means we have to modify the `initialize()` method to make them invisible at first (lines 80 – 81 in Figure 43)

```
67  @FXML
68  public void initialize() {
69      colMediaId.setCellValueFactory(
70              new PropertyValueFactory<Media, Integer>("id"));
71      colMediaTitle.setCellValueFactory(
72              new PropertyValueFactory<Media, String>("title"));
73      colMediaCategory.setCellValueFactory(
74              new PropertyValueFactory<Media, String>("category"));
75      colMediaCost.setCellValueFactory(
76              new PropertyValueFactory<Media, Float>("cost"));
77      if(cart.getItemsOrdered() != null )
78          tblMedia.setItems(cart.getItemsOrdered());
79
80      btnPlay.setVisible(false);
81      btnRemove.setVisible(false);
82
83      tblMedia.getSelectionModel().selectedItemProperty().addListener(new ChangeListener<Media>() {
84          @Override
85          public void changed(ObservableValue<? extends Media> observable, Media oldValue, Media newValue) {
86              updateButtonBar(newValue);
87          }
88      });
89  }
```

Figure 43. Modified initialize() method

Put some code at the end of the `initialize()` method to add a `ChangeListener` to the `TableView`'s `selectedItem` property (lines 83 – 88 in Figure 43). Here, we create an anonymous inner class for the `ChangeListener`. All `ChangeListeners` must implement the `changed()` method. Whenever a selected item in the `TableView` is changed, the method `changed()` is called. Here, we check to make sure the `newValue` is not `null` (the user didn't just unselect) and call the `updateButtonBar()` method (Figure 44)

```
void updateButtonBar(Media media) {
    if (media == null) {
        btnPlay.setVisible(false);
        btnRemove.setVisible(false);
    }
    else {
        btnRemove.setVisible(true);
        if(media instanceof Playable) {
            btnPlay.setVisible(true);
        }
        else {
            btnPlay.setVisible(false);
        }
    }
}
```

Figure 44. Source code of updateButtonBar()

6.5 *Deleting a media*

Next, we will implement the event handling for the "Remove" button. Please add a method name to the onAction property of the button in Scene Builder. You can refer to the event-handling code below:

```
@FXML
void btnRemovePressed(ActionEvent event) {
    Media media = tblMedia.getSelectionModel().getSelectedItem();
    cart.removeMedia(media);

}
```

Figure 45. Handle remove media

Note that we don't need to invoke an update for the `TableView` because it can already observe the changes through the `ObservableList` and update its display.

6.6 *Filter items in cart – `FilteredList`*

This exercise is optional (full credit can still be given for this lab without doing this exercise), but you can do it for extra credit.

We will implement a filter that is re-applied every time the user makes a change in the filter text field. To do this, again, we need references to the text field where the user inputs the filter string, and the two radio buttons (to determine what criteria are being used to filter).

Similar to the above, please add the fx:id property for the components in SceneBuilder and create three corresponding attributes in the controller:

- The TextField: **tfFilter**
- The RadioButton "By ID": **radioBtnFilterId**
- The RadioButton "By Title": **radioBtnFilterTitle**

At the end of the `initialize()` method, put some code to add a `ChangeListener` to the `TextField`'s text property (illustrated in Figure 46):

```
tfFilter.textProperty().addListener(new ChangeListener<String>() {
    @Override
    public void changed(ObservableValue<? extends String> observable, String oldValue,
            String newValue) {
        showFilteredMedia(newValue);
    }
});
```

Figure 46. Adding ChangListener for tfFilter in initialize()

Please implement by yourself the `showFilteredMedia()` method.

Hint: You might need to change the source code in previous exercises. Wrap the `ObservableList` in a `FilteredList` and set a new `Predicate` for the `FilteredList` each time you need to apply a new filter.

7 Switch Screen between Store and Cart

In JavaFX, an application can only have one stage but that stage can have one or several scenes. Therefore, can create multiple scenes for a given JavaFX application and consider each screen of the application as a scene. This helps us easily switch between multiple screens.

The following code is used to switch from Store Screen to Cart Screen when user click on the button View Cart in Store Screen:

```
62⊖    @FXML
63     void btnViewCartPressed(ActionEvent event) {
64         try {
65             final String CART_FXML_FILE_PATH = "/hust/soict/globalict/aims/screen/customer/view/Cart.fxml";
66             FXMLLoader fxmlLoader = new FXMLLoader(getClass().getResource(CART_FXML_FILE_PATH));
67                 fxmlLoader.setController(new CartController(store, cart));
68                 Parent root = fxmlLoader.load();
69                 Stage stage = (Stage)((Node) event.getSource()).getScene().getWindow();
70                 stage.setScene(new Scene(root));
71                 stage.setTitle("Cart");
72                 stage.show();
73         } catch (IOException e) {
74             e.printStackTrace();
75         }
76     }
77 }
```

Figure 47. Source code switch from Store Screen to Cart Screen

This code is placed in *`ViewStoreController`* class created earlier. Make sure that you add the On Action property for the View Cart button.

Note: You may need to add the `Cart cart` attribute to the ViewStoreController class and modify its constructor to pass both `Store store` and `Cart cart` to the constructor.

Explain the code:

- Lines 65-68: load the fxml file of the Cart Screen
- Line 69: The `getSource()` method returns an object on which the event initially occurred. Call `getScene().getWindow()` on the node on which the action occurred to get the "current" window.
- Line 70-72: Set new scene for the current stage

The code to switch from Cart Screen to Store Screen is similar.

8 Complete the Aims GUI application

Complete the remaining UI of Aims to make a functioning GUI application

- Store Screen:
  - "Play" Button
  - "Add to cart" Button
- Cart Screen:
  - "View Store" button
  - "Play" Button
  - "Place order" Button
  - The total cost Label - should update along with changes in the current cart (add/remove).
  - Filter (Search)

9 Check all the previous source codes to catch/handle/delegate runtime exceptions

Review all methods, classes in AimsProject, catch/handle or delegate all exceptions if necessary.
The exception delegation mechanism is especially helpful for constructors so that no object is created if there is any violation of the requirement/constraints.

Hint: In Aims Project, we can apply exception handling to *validate data constraints* such as non-negative price, to *validate policies* like the restriction of the number of orders, and to handle *unexpected interactions*, e.g., user try to remove an author while the author is not listed.

For example, the following piece of code illustrates how to control the number of items in the cart with exception.

```
public void addMedia(Media m) throws LimitExceededException {
    if(itemsOrdered.size() < MAX_NUMBERS_ORDERED) {
        //TODO add media into cart
    }
    else {
        throw new LimitExceededException("ERROR: The number of "
                + "media has reached its limit");
    }
}
```

Figure 48. Sample exception handling code

10 Create a class which inherits from **`Exception`**

The **`PlayerException`** class represents an exception that will be thrown when an exceptional condition occurs during the playing of a media in your **`AimsProject`**.

10.1 Create new class named `PlayerException`

- Enter the following specifications in the New Java Class dialog:
  - Name: **`PlayerException`**
  - Package: **`hust.soict.[globalict||dsai].aims.exception`**
  - Access Modifier: **`public`**
  - Superclass: **`java.lang.Exception`**
  - Constructor from Superclass: checked
  - **`public static void main(String [] args)`**: do not check
  - All other boxes: do not check
- Finish

10.2 Raise the `PlayerException` in the `play()` method

- Update **`play()`** method in **`DigitalVideoDisc`** and **`Track`**
  - For each of **`DigitalVideoDisc`** and **`Track`**, update the **`play()`** method to first check the object's length using **`getLength()`** method. If the length of the **`Media`** is less than or equal to zero, the **`Media`** object cannot be played.

- At this point, you should output an error message using **`System.err.println()`** method and the **`PlayerException`** should be raised.

- The example of codes and results for the **`play`**() of **`DigitalVideoDisc`** are illustrated in the following figures.

```
public void play() throws PlayerException {
	if (this.getLength() > 0) {
		// TODO Play DVD as you have implemented
	} else {
		throw new PlayerException("ERROR: DVD length is non-positive!");
	}
}
```

Figure 49. Sample code for method play() of DigitalVideoDisc

- Save your changes and make the same with the `play()` method of `Track`.

10.3 Update `play()` in the `Playable` interface

- Change the method signature for the **`Playable`** interface's **`play()`** method to include the throws **`PlayerException`** keywords.

10.4 Update play( ) in CompactDisc

- The **`play`**() method in the **`CompactDisc`** is more interesting because not only it is possible for the **`CompactDisc`** to have an invalid **`length`** of 0 or less, but it is also possible that as it iterates through the tracks to play each one, there may have a track of length 0 or less

- First update the **`play`**() method in **`CompactDisc`** class to check the length using **`getLength`**() method as you did with **`DigitalVideoDisc`**

- Raise the **`PlayerException`**. Be sure to change the method signature to include **`throws PlayerException`** keywords.

- Update the **`play()`** method to catch a **`PlayerException`** raised by each **`Track`** using block **`try-catch.`**

The code example is shown as follows.

```
public void play() throws PlayerException{
	if(this.getLength() > 0) {
		// TODO Play all tracks in the CD as you have implemented
		java.util.Iterator iter = tracks.iterator();
		Track nextTrack;
		while(iter.hasNext()) {
			nextTrack = (Track) iter.next();
			try {
				nextTrack.play();
			}catch(PlayerException e) {
				throw e;
			}
		}
	}else {
		throw new PlayerException("ERROR: CD length is non-positive!");
	}
}
```

Figure 50. Sample code for method play() of CompactDisc

- You should modify the above source code so that if any track in a **CD** can't play, it throws a **PlayerException** exception.

11 Update the **Aims** class

- The **Aims** class must be updated to handle any exceptions generated when the **play()** methods are called. What happens when you don't update for them to catch?
- Try to use **try-catch** block when you call the **play()** method of **Media's** objects.

With all these steps, you have practiced with User-defined Exception (**PlayerException**), **try-catch** block and also **throw**. The **try-catch** block is used in the main method of class **Aims.java** and in the **play()** method of the **CompactDisc.java**. Print all information of the exception object, e.g. **getMessage()**, **toString()**, **printStackTrace()**, display a dialog box to the user with the content of the exception.

The example of codes and results for the **play**() of **DigitalVideoDisc** in **Swing** are illustrated in the following figure.

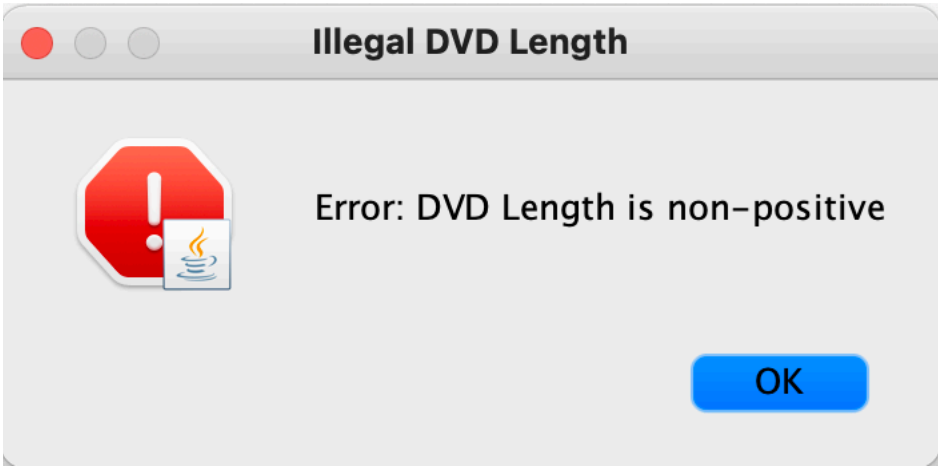


Figure 51. Swing dialog showing error

12 Modify the `equals()` method and `compareTo()` method of Comparable for Media class

- Two medias are equals if they have the same **`title`** and **`cost`**

- Please remember to check for **`NullPointerException`** and **`ClassCastException`** if applicable.

You may use **`instanceof`** operator to check if an object is an instance of a **`ClassType`**.

13 Reading Document

Please read the following links for better understanding.

- Exception-handling basics: https://developer.ibm.com/tutorials/j-perry-exceptions/
- Basic guidelines: Although the examples are in C++, the ideas are important. https://docs.microsoft.com/en-us/cpp/cpp/errors-and-exception-handling-modern-cpp?view=vs-2019#basic-guidelines

14 Exercises: Hierarchical tree diagram

- Make an exception hierarchical tree for all self-defined exceptions in Aims Project. Use the class diagram in Astah to draw this tree, export it as a png file, and save them in the design directory.