Object-Oriented Programming

Lecturer:
Teaching Assistant:

Lab 05: GUI Programming

In this lab, you will practice with:

- Create simple GUI applications with Swing
- Create simple GUI applications with JavaFX
- Convert the Aims Project from the console/command-line (CLI) application to the GUI one
- Use Exception in your program

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.
- **10PM five days after the class**: for this deadline, you should include the **source code** of all sections of this lab, into a branch namely "**release/lab05**" 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 AIMS project.

Each student is expected to turn in his or her own 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. Swing components

Note: For this exercise, you will create a new Java project named `GUIProject`, and put all your source code in a package called "**hust.soict.globalict.swing**" (for ICT) or "**hust.soict.dsai.swing**" (for DS & AI).

In this exercise, we revisit the elements of the Swing API and compare them with those of AWT through implementing the same mini-application using the two libraries. The application is an accumulator which accumulates the values entered by the user and display the sum.

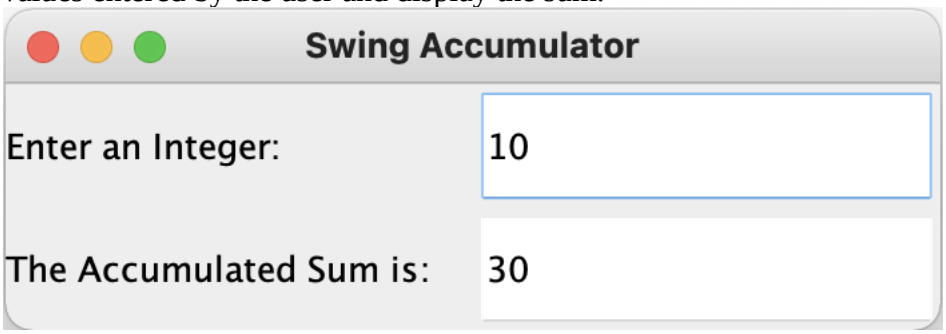

Figure 1. SwingAccumulator

1.1. `AWTAccumulator`

1.1.1. Create class `AWTAccumulator` with the source code as below

```
 6 public class AWTAccumulator extends Frame {
 7    private TextField tfInput;
 8    private TextField tfOutput;
 9    private int sum = 0;        // Accumulated sum, init to 0
10
11    // Constructor to setup the GUI components and event handlers
12    public AWTAccumulator() {
13       setLayout(new GridLayout(2, 2));
14
15       add(new Label("Enter an Integer: "));
16
17       tfInput = new TextField(10);
18       add(tfInput);
19       tfInput.addActionListener(new TFInputListener());
20
21       add(new Label("The Accumulated Sum is: "));
22
23       tfOutput = new TextField(10);
24       tfOutput.setEditable(false);
25       add(tfOutput);
26
27       setTitle("AWT Accumulator");
28       setSize(350, 120);
29       setVisible(true);
30    }
31
32    public static void main(String[] args) {
33       new AWTAccumulator();
34    }
35
36    private class TFInputListener implements ActionListener {
37       @Override
38       public void actionPerformed(ActionEvent evt) {
39          int numberIn = Integer.parseInt(tfInput.getText());
40          sum += numberIn;
41          tfInput.setText("");
42          tfOutput.setText(sum + "");
43       }
44    }
45 }
```

Figure 2. Source code of AWTAccumulator

1.1.2. Explanation

- In AWT, the top-level container is `Frame`, which is inherited by the application class.
- In the constructor, we set up the GUI components in the `Frame` object and the event-handling:
  - In line 13, the layout of the frame is set as `GridLayout`

- In line 15, we add the first component to our `Frame`, an anonymous `Label`
- In line 17-19, we add a `TextField` component to our `Frame`, where the user will enter values. We add a listener which takes this `TextField` component as the source, using a named inner class.
- In line 21, we add another anonymous `Label` to our `Frame`
- In line 23 – 25, we add a `TextField` component to our `Frame`, where the accumulated sum of entered values will be displayed. The component is set to read-only in line 24.
- In line 27 – 29, the title & size of the `Frame` is set, and the `Frame` visibility is set to true, which shows the `Frame` to us.

- In the listener class (line 36 - 44), the `actionPerformed()` method is implemented, which handles the event when the user hit "Enter" on the source `TextField`.
  - In line 39-42, the entered number is parsed, added to the sum, and the output `TextField's` text is changed to reflect the new sum.
- In the `main()` method, we invoke the `AWTAccumulator` constructor to set up the GUI

1.2. `SwingAccumulator`

1.2.1. Create class `SwingAccumulator` with the source code as below:

```
 8 public class SwingAccumulator extends JFrame {
 9    private JTextField tfInput;
10    private JTextField tfOutput;
11    private int sum = 0;        // Accumulated sum, init to 0
12
13    // Constructor to setup the GUI components and event handlers
14    public SwingAccumulator() {
15       Container cp = getContentPane();
16       cp.setLayout(new GridLayout(2, 2));
17
18       cp.add(new JLabel("Enter an Integer: "));
19
20       tfInput = new JTextField(10);
21       cp.add(tfInput);
22       tfInput.addActionListener(new TFInputListener());
23
24       cp.add(new JLabel("The Accumulated Sum is: "));
25
26       tfOutput = new JTextField(10);
27       tfOutput.setEditable(false);
28       cp.add(tfOutput);
29
30       setTitle("Swing Accumulator");
31       setSize(350, 120);
32       setVisible(true);
33    }
34
35    public static void main(String[] args) {
36       new SwingAccumulator();
37    }
38
39    private class TFInputListener implements ActionListener {
40       @Override
41       public void actionPerformed(ActionEvent evt) {
42          int numberIn = Integer.parseInt(tfInput.getText());
43          sum += numberIn;
44          tfInput.setText("");
45          tfOutput.setText(sum + "");
46       }
47    }
48 }
```

Figure 3. Source code of SwingAccumulator

1.2.2. Explanation

- In Swing, the top-level container is `JFrame` which is inherited by the application class.

- In the constructor, we set up the GUI components in the `JFrame` object and the event-handling:
  - Unlike AWT, the `JComponents` shall not be added onto the top-level container (e.g., `JFrame`, `JApplet`) directly because they are lightweight components. The `JComponents` must be added onto the so-called content-pane of the top-level container. Content-pane is in fact a `java.awt.Container` that can be used to group and layout components.
  - In line 15, we get the content-pane of the top-level container.
  - In line 16, the layout of the content-pane is set as `GridLayout`
  - In line 18, we add the first component to our content-pane, an anonymous `JLabel`
  - In line 20-22, we add a `JTextField` component to our content-pane, where the user will enter values. We add a listener which takes this `JTextField` component as the source.
  - In line 24, we add another anonymous `JLabel` to our content-pane
  - In line 26 – 28, we add a `JTextField` component to our content-pane, where the accumulated sum of entered values will be displayed. The component is set to read-only in line 27.
  - In line 30 – 32, the title & size of the `JFrame` is set, and the Frame visibility is set to true, which shows the `JFrame` to us.
- In the listener class (line 39 - 47), the code for event-handling is exactly like the `AWTAccumulator`.
- In the `main()` method, we invoke the `SwingAccumulator` constructor to set up the GUI

1.3. Compare Swing and AWT elements

Programming with AWT and Swing is quite similar (similar elements including container/components, event-handling). However, there are some differences that you need to note:

- The top-level containers in Swing and AWT
- The class name of components in AWT and corresponding class's name in Swing

2. Organizing Swing components with Layout Managers

Note: For this exercise, you will continue using `GUIProject`, and put all your source code in the package "**hust.soict.globalict.swing**" (for ICT) or "**hust.soict.dsai.swing**" (for DS & AI).

In Swing, there are two groups of GUI classes, the containers and the components. We have worked with several component classes in previous exercises. Now, we will investigate more on the containers.

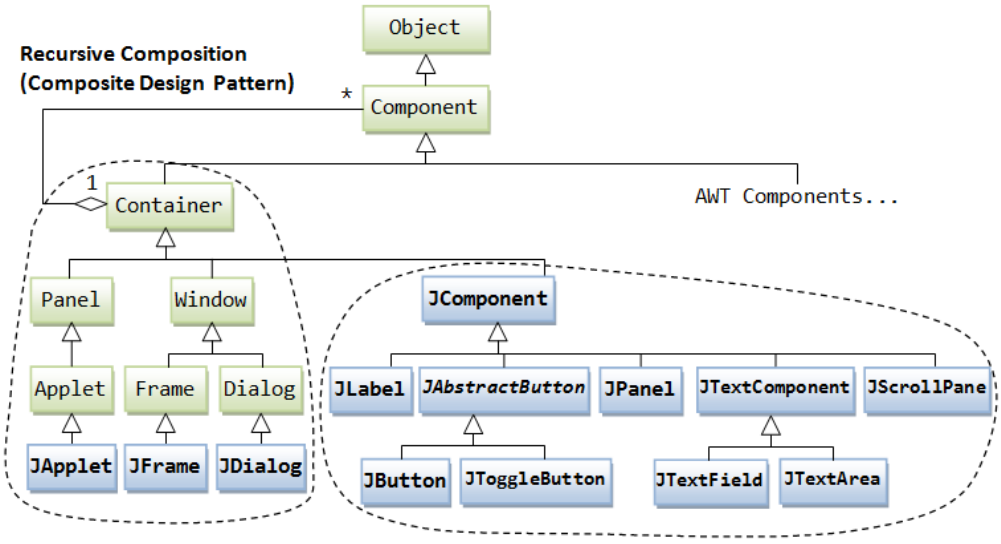

Figure 4. AWT and Swing elements

2.1. Swing top-level and secondary-level containers

A container is used to hold components. A container can also hold containers because it is a (subclass of) component. Swing containers are divided into top-level and secondary-level containers:

- Top-level containers:
  - `JFrame`: used for the application's main window (with an icon, a title, minimize/maximize/close buttons, an optional menu-bar, and a content-pane)
  - `JDialog`: used for secondary pop-up window (with a title, a close button, and a content-pane).
  - `JApplet`: used for the applet's display-area (content-pane) inside a browser's window
- Secondary-level containers can be used to group and layout relevant components (most commonly used is `JPanel`)

2.2. Using JPanel as secondary-level container to organize components

2.2.1. Create class `NumberGrid`

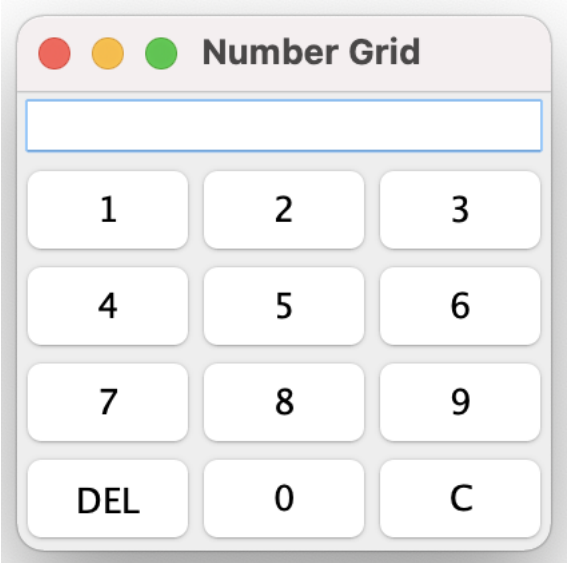

Figure 5. NumberGrid

This class allows us to input a number digit-by-digit from a number grid into a text field display. We can also delete the latest digit or delete the entire number and start over.

```
16  public class NumberGrid extends JFrame{
17      private JButton[] btnNumbers = new JButton[10];
18      private JButton btnDelete, btnReset;
19      private JTextField tfDisplay;
20
21      public NumberGrid() {
22
23          tfDisplay = new JTextField();
24          tfDisplay.setComponentOrientation(
25                  ComponentOrientation.RIGHT_TO_LEFT);
26
27          JPanel panelButtons = new JPanel(new GridLayout(4, 3));
28          addButtons(panelButtons);
29
30          Container cp = getContentPane();
31          cp.setLayout(new BorderLayout());
32          cp.add(tfDisplay, BorderLayout.NORTH);
33          cp.add(panelButtons, BorderLayout.CENTER);
34
35          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
36          setTitle("Number Grid");
37          setSize(200, 200);
38          setVisible(true);
39      }
```

Figure 6. NumberGrid source code(1)

The class has several attributes:

- The `btnNumbers` array for the digit buttons
- The `btnDelete` for the DEL button
- The `btnReset` for the C button
- The `tfDisplay` for the top display

In the constructor, we add two components to the content pane of the `JFrame`:

- A `JTextField` for the display text field
- A `JPanel`, which will group all of the buttons and put them in a grid layout

2.2.2. Adding buttons

We add the buttons to the `panelButtons` in the `addButtons()` method:

```
40    void addButtons(JPanel panelButtons) {
41        ButtonListener btnListener = new ButtonListener();
42        for(int i = 1; i <= 9; i++) {
43            btnNumbers[i] = new JButton(""+i);
44            panelButtons.add(btnNumbers[i]);
45            btnNumbers[i].addActionListener(btnListener);
46        }
47
48        btnDelete = new JButton("DEL");
49        panelButtons.add(btnDelete);
50        btnDelete.addActionListener(btnListener);
51
52        btnNumbers[0] = new JButton("0");
53        panelButtons.add(btnNumbers[0]);
54        btnNumbers[0].addActionListener(btnListener);
55
56        btnReset = new JButton("C");
57        panelButtons.add(btnReset);
58        btnReset.addActionListener(btnListener);
59    }
```

Figure 7. NumberGrid source code(2)

The buttons share the same listener of the `ButtonListener` class, which is a named inner class.

2.2.3. Complete inner class `ButtonListener`

Your task is to complete the implementation of the `ButtonListener` class below:

```
71    private class ButtonListener implements ActionListener{
72        @Override
73        public void actionPerformed(ActionEvent e) {
74            String button = e.getActionCommand();
75            if(button.charAt(0) >= '0' && button.charAt(0) <= '9') {
76                tfDisplay.setText(tfDisplay.getText() + button);
77            }
78            else if (button.equals("DEL")) {
79                //handles the "DEL" case
80            }
81            else {
82                //handles the "C" case
83            }
84
85        }
86
87    }
```

Figure 8. NumberGrid source code(3)

In the `actionPerformed()` method, we will handle the button pressed event. Since we have many sources, we need to determine which source is firing the event (which button is pressed) and handle each case accordingly (change the text of the display text field). Here, we have three cases:

- A digit button: a digit is appended to the end
- DEL button: delete the last digit
- C button: clears all digits

The code for the first case is there for reference, you need to implement by yourself the remaining two cases.

3. Create a graphical user interface for AIMS with Swing

For the AIMS application, we will implement three screens:

- The "View Store" screen using Swing
- The "View Cart" screen using JavaFX
- The "Update Store" screen using either Swing or JavaFX depending on you

In this exercise, we will make the first screen of the Aims application, the View Store Screen

Put the source code in this exercises in the AimsProject's "**hust.soict.globalict.aims.screen**" package (for ICT) or the "**hust.soict.dsai.aims.screen**" package (for DS&AI).

3.1. View Store Screen

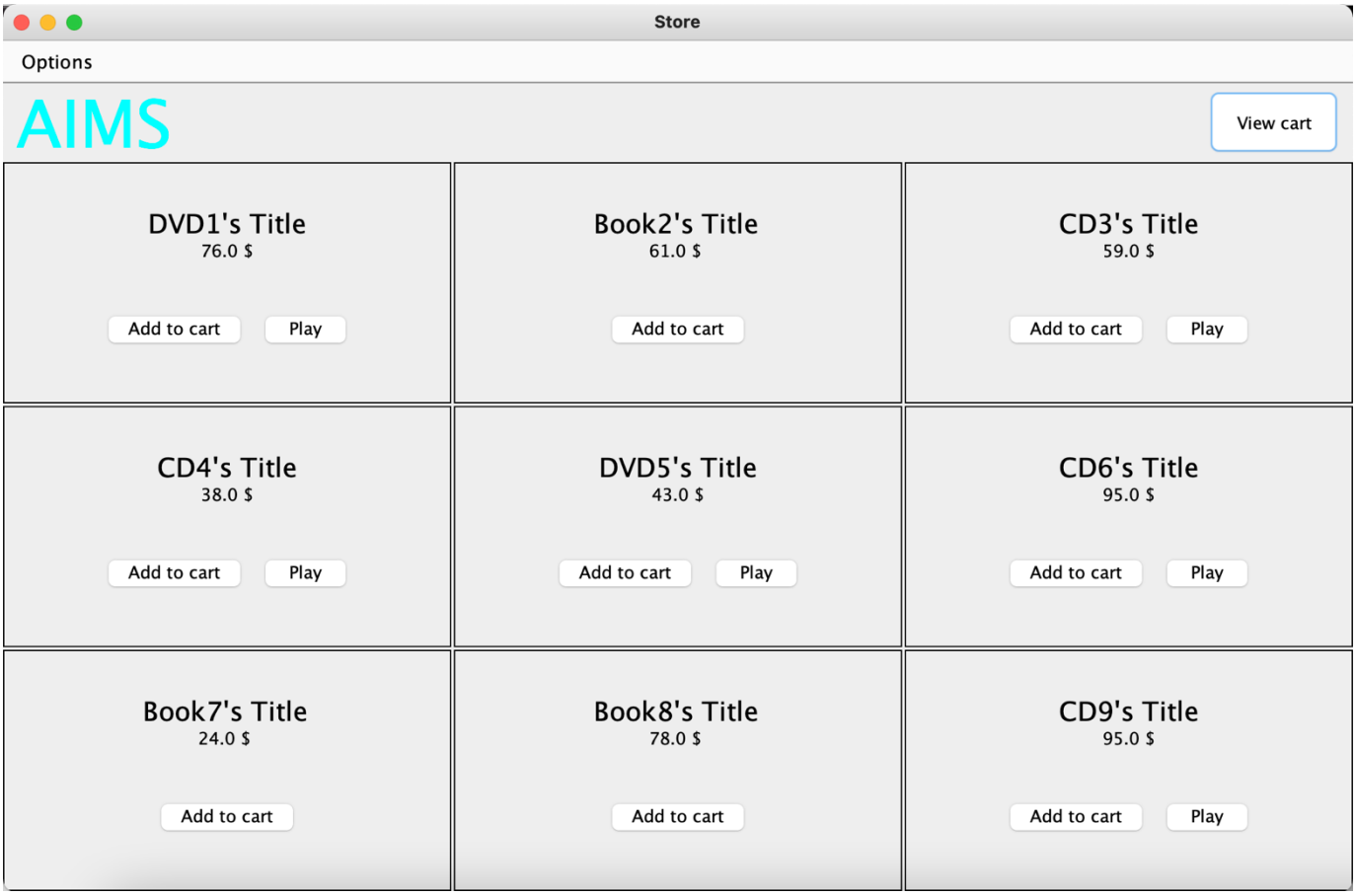

Figure 9. View Store Screen

For the View Store Screen, we will use the `BorderLayout`:

In the NORTH component, there will be the menu bar and the header

In the CENTER component, there will be a panel which uses the `GridLayout`, each cell is an item in the store.

3.1.1. Create the `StoreScreen` class

```
public class StoreScreen extends JFrame{
    private Store store;
```

Figure 10. Declaration of StoreScreen class

This will be our View Store Screen.

Declare one attribute in the `StoreScreen` class: `Store store`. This is because we need information on the items in the store to display them

3.1.2. The NORTH component

Create the method createNorth(), which will create our NORTH component:

```
51    JPanel createNorth() {
52        JPanel north = new JPanel();
53        north.setLayout(new BoxLayout(north, BoxLayout.Y_AXIS));
54        north.add(createMenuBar());
55        north.add(createHeader());
56        return north;
57    }
```

Figure 11. createNorth() source code

Create the method `createMenuBar()`:

```
59    JMenuBar createMenuBar() {
60
61        JMenu menu = new JMenu("Options");
62
63        JMenu smUpdateStore = new JMenu("Update Store");
64        smUpdateStore.add(new JMenuItem("Add Book"));
65        smUpdateStore.add(new JMenuItem("Add CD"));
66        smUpdateStore.add(new JMenuItem("Add DVD"));
67
68        menu.add(smUpdateStore);
69        menu.add(new JMenuItem("View store"));
70        menu.add(new JMenuItem("View cart"));
71
72
73        JMenuBar menuBar = new JMenuBar();
74        menuBar.setLayout(new FlowLayout(FlowLayout.LEFT));
75        menuBar.add(menu);
76
77        return menuBar;
78    }
```

Figure 12. createMenuBar() source code

The resulting menu bar will look something like this:

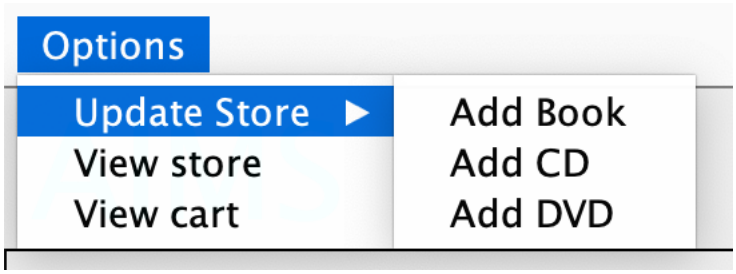

Figure 13. Resulting menu on menu bar

Create the method `createHeader()`:

```
79    JPanel createHeader() {
80
81        JPanel header = new JPanel();
82        header.setLayout(new BoxLayout(header, BoxLayout.X_AXIS));
83
84        JLabel title = new JLabel("AIMS");
85        title.setFont(new Font(title.getFont().getName(), Font.PLAIN, 50));
86        title.setForeground(Color.CYAN);
87
88        JButton cart = new JButton("View cart");
89        cart.setPreferredSize(new Dimension(100, 50));
90        cart.setMaximumSize(new Dimension(100, 50));
91
92        header.add(Box.createRigidArea(new Dimension(10, 10)));
93        header.add(title);
94        header.add(Box.createHorizontalGlue());
95        header.add(cart);
96        header.add(Box.createRigidArea(new Dimension(10, 10)));
97
98        return header;
99    }
```

Figure 14. createHeader() source code

3.1.3. The `CENTER` component

```
101    JPanel createCenter() {
102
103        JPanel center = new JPanel();
104        center.setLayout(new GridLayout(3, 3, 2, 2));
105
106        ArrayList<Media> mediaInStore = store.getItemsInStore();
107        for (int i = 0; i < 9; i++) {
108            MediaStore cell = new MediaStore(mediaInStore.get(i));
109            center.add(cell);
110        }
111
112
113        return center;
114    }
```

Figure 15. createCenter() source code

Here, we see that each cell is an object of class `MediaStore`, which represents the GUI element for a `Media` in the Store Screen.

3.1.4. The `MediaStore` class

Here, since the `MediaStore` is a GUI element, it extends the `JPanel` class. It has one attribute: `Media media`.

```
17 public class MediaStore extends JPanel{
18     private Media media;
19     public MediaStore(Media media){
20 
21         this.media = media;
22         this.setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
23 
24         JLabel title = new JLabel(media.getTitle());
25         title.setFont(new Font(title.getFont().getName(), Font.PLAIN, 20));
26         title.setAlignmentX(CENTER_ALIGNMENT);
27 
28 
29         JLabel cost = new JLabel(""+media.getCost()+" $");
30         cost.setAlignmentX(CENTER_ALIGNMENT);
31 
32         JPanel container = new JPanel();
33         container.setLayout(new FlowLayout(FlowLayout.CENTER));
34 
35         container.add(new JButton("Add to cart"));
36         if(media instanceof Playable) {
37             container.add(new JButton("Play"));
38         }
39 
40         this.add(Box.createVerticalGlue());
41         this.add(title);
42         this.add(cost);
43         this.add(Box.createVerticalGlue());
44         this.add(container);
45 
46         this.setBorder(BorderFactory.createLineBorder(Color.BLACK));
47     }
48 }
```

Figure 16. MediaStore source code

Note how the code checks if the `Media` implements the `Playable` interface to create a “Play” button.

3.1.5. Putting it all together

Finally, we have all the component methods to use in the constructor of `StoreScreen`:

```
35     public StoreScreen(Store store) {
36         this.store = store;
37         Container cp = getContentPane();
38         cp.setLayout(new BorderLayout());
39 
40         cp.add(createNorth(), BorderLayout.NORTH);
41         cp.add(createCenter(), BorderLayout.CENTER);
42 
43         setVisible(true);
44         setTitle("Store");
45         setSize(1024, 768);
46     }
```

Figure 17. StoreScreen constructor source code

3.2. Adding more user interaction

We have successfully set up all the components for our store, but they are just static – buttons and menu items don't respond when being clicked. Now, it's your task to implement the handling of the event when the user interacts with:

- The buttons on `MediaHome`:
  - When user clicks on the "Play" button, the Media should be played in a dialog window. You can use `JDialog` here.
  - When user clicks on the "Add to cart" button, the `Media` should be added to the cart.

Note: The GUI of the "View cart" & "Update Store" functionality will be implemented in the next exercises. For now, it should suffice to still use the console interface for them.

4. JavaFX API

Note: For this exercise, 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 JDK version after 1.8***.

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

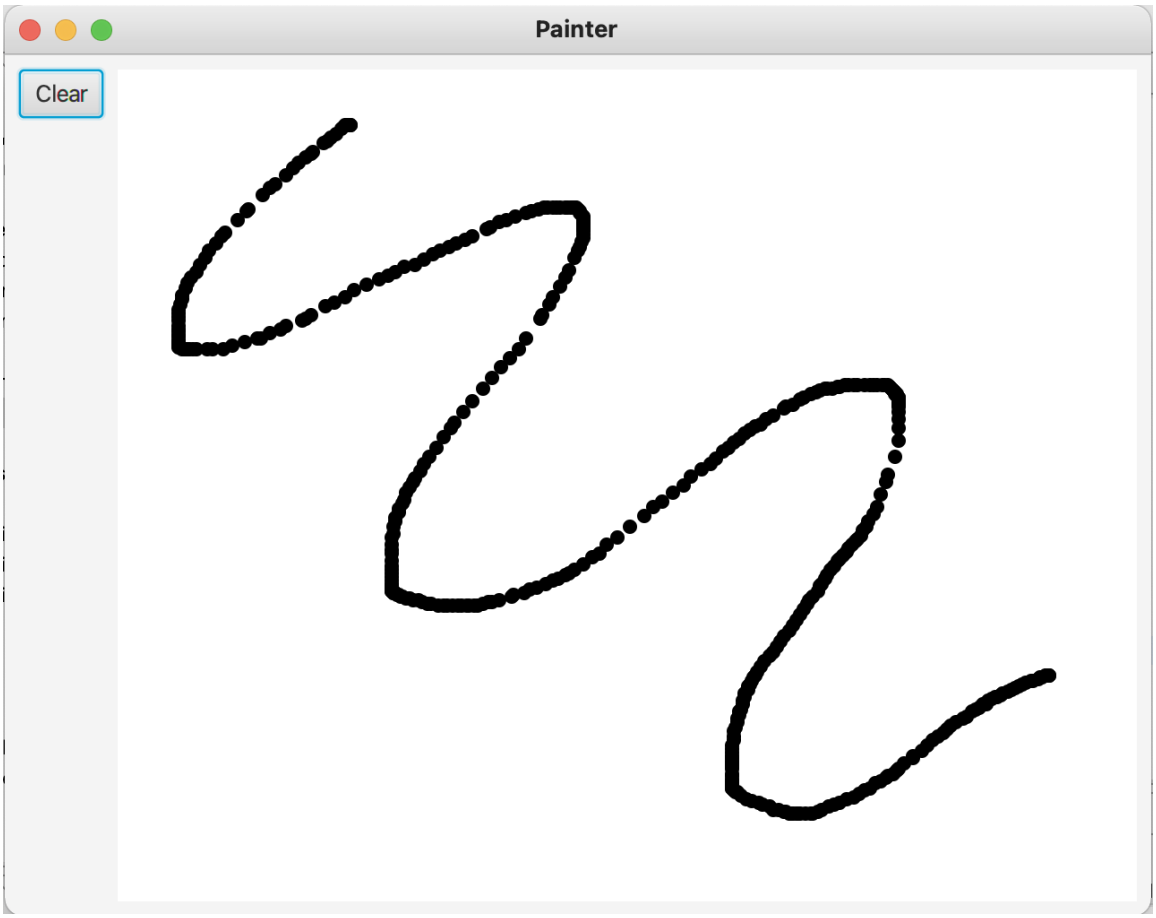

Figure 18. Painter app

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`).

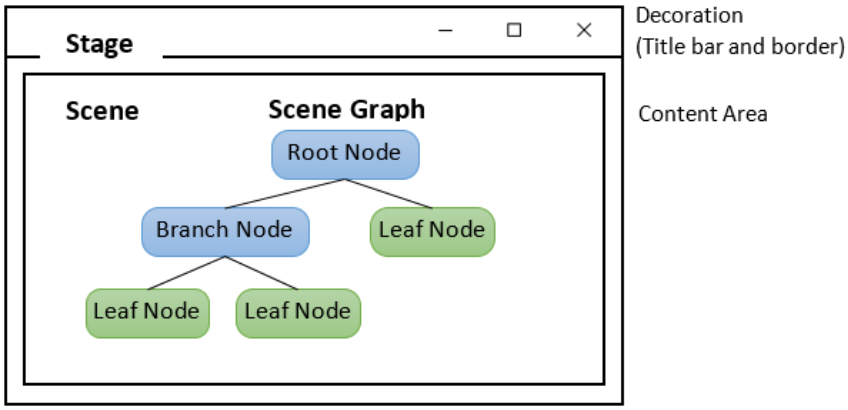

Figure 19. Structure of JavaFX application

Like any other JavaFX applications, 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 the 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 add it to the stage.

4.1. Create the FXML file

4.1.1. 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 20.

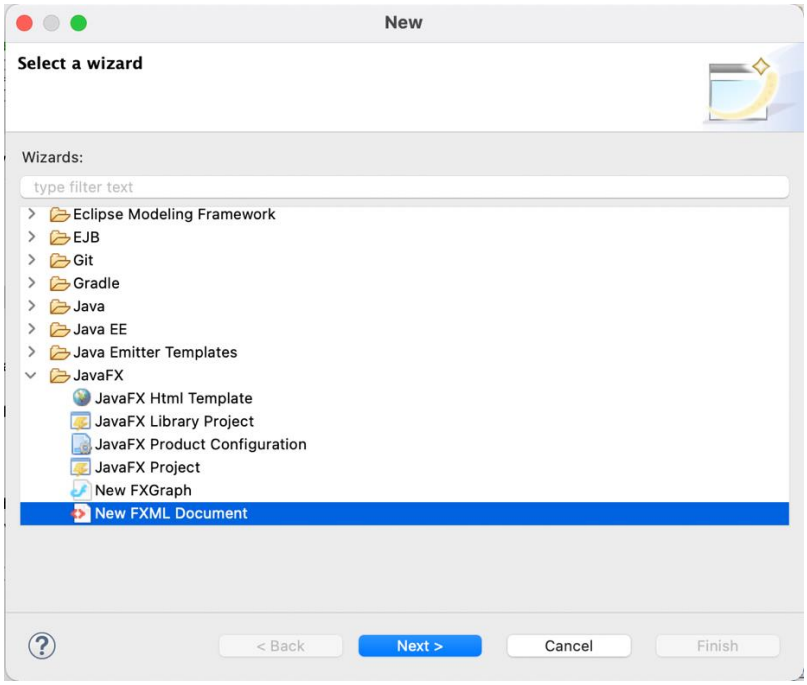

Figure 20. Create a new FXML in Eclipse(1)

Name the file "`Painter`" and choose `BorderPane` as the root element as in Figure 21

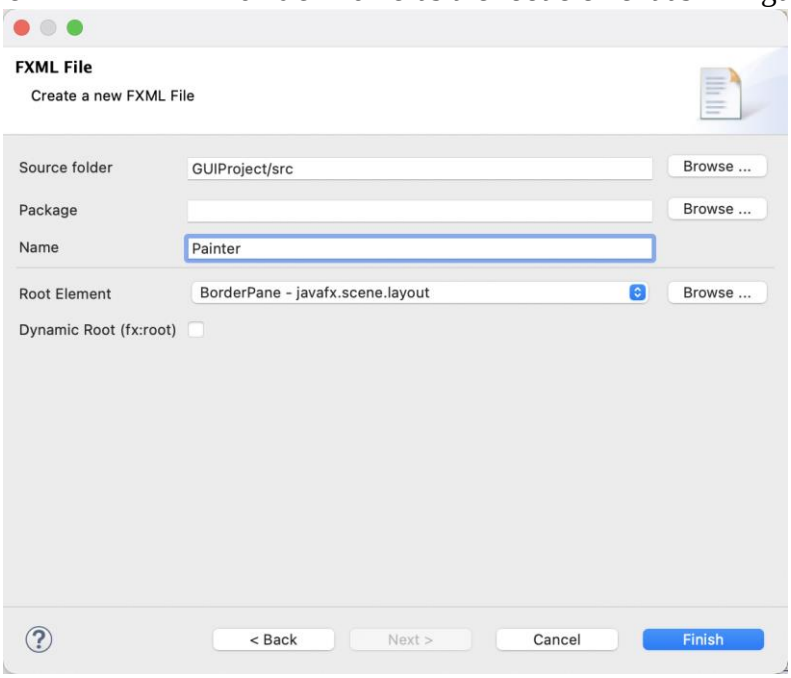

Figure 21. Create a new FXML in Eclipse(2)

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

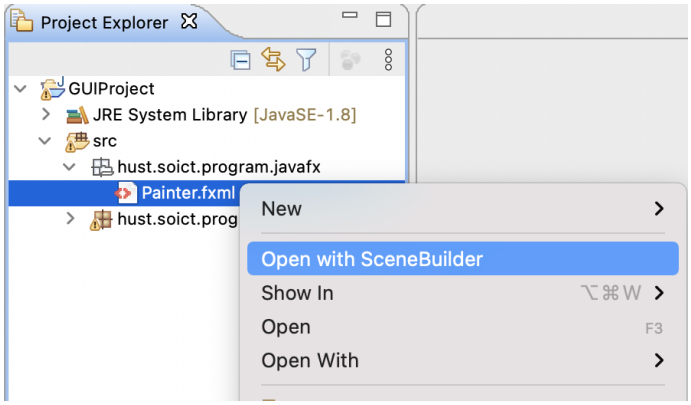

Figure 22. Open FXML with SceneBuilder from Eclipse

4.1.2. 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.

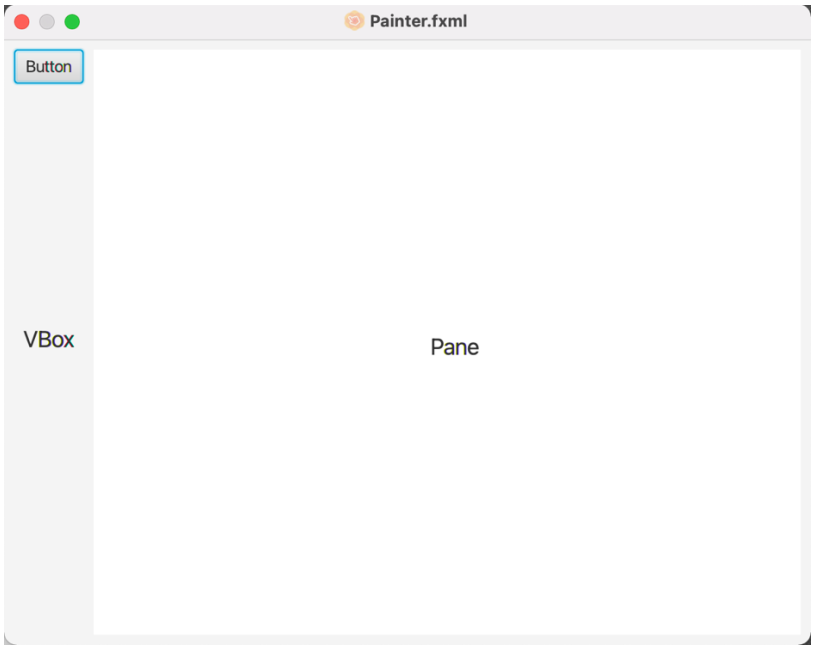

Figure 23. Target interface

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

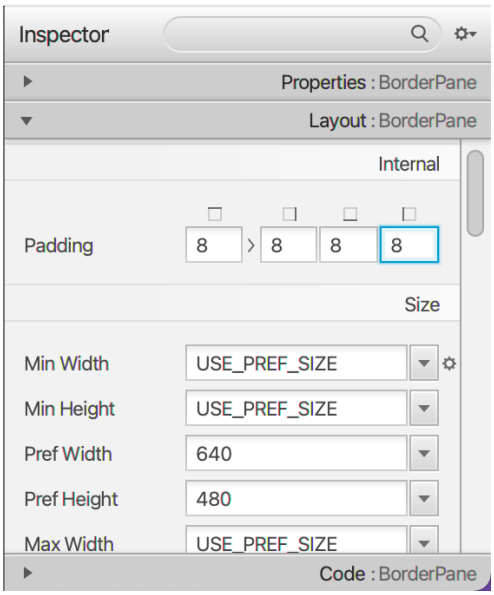

Figure 24. Configuring the BorderPane

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.

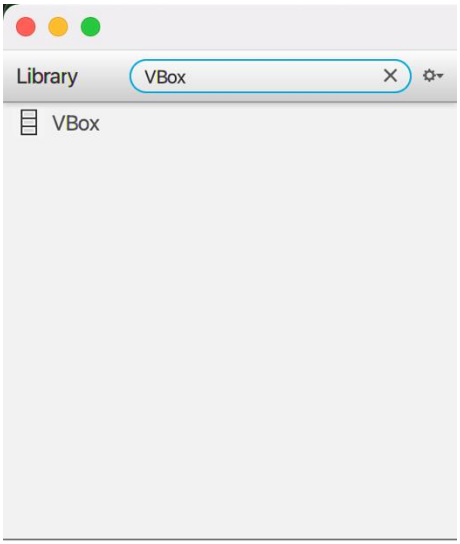

Figure 25. Get VBox in the Library menu

- 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)

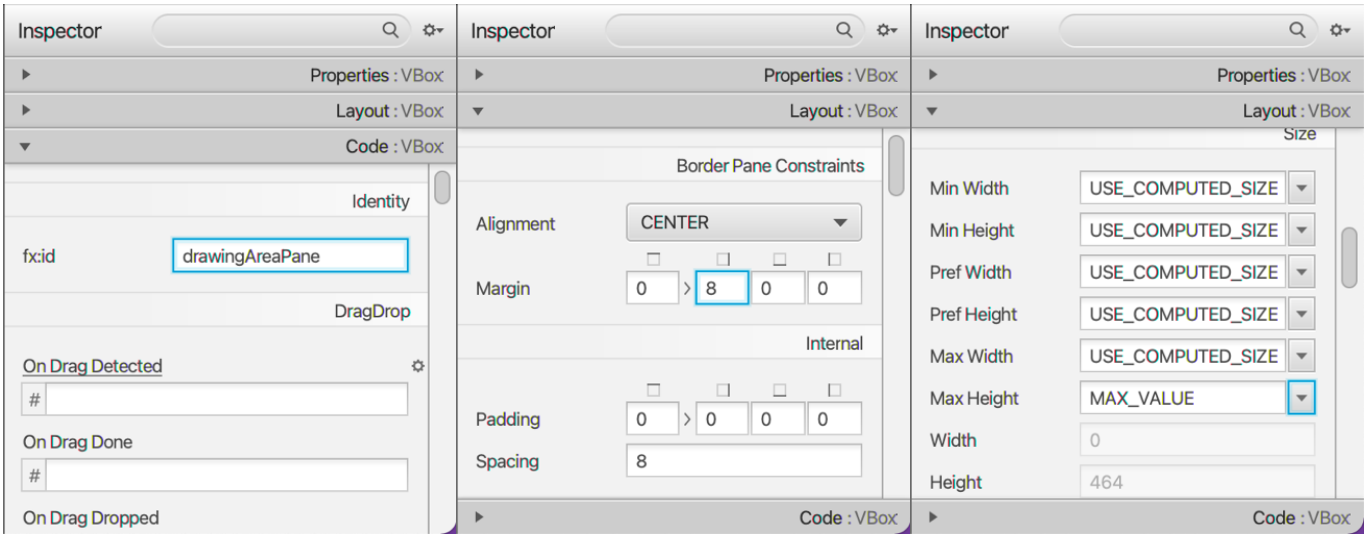

Figure 26. Configuring the VBox

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.

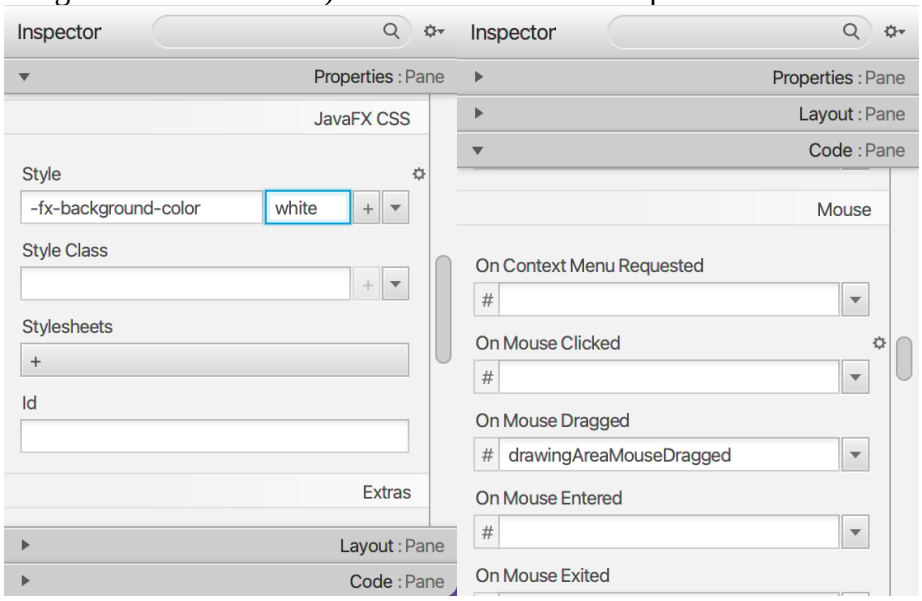

Figure 27. Configuring the Pane

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.

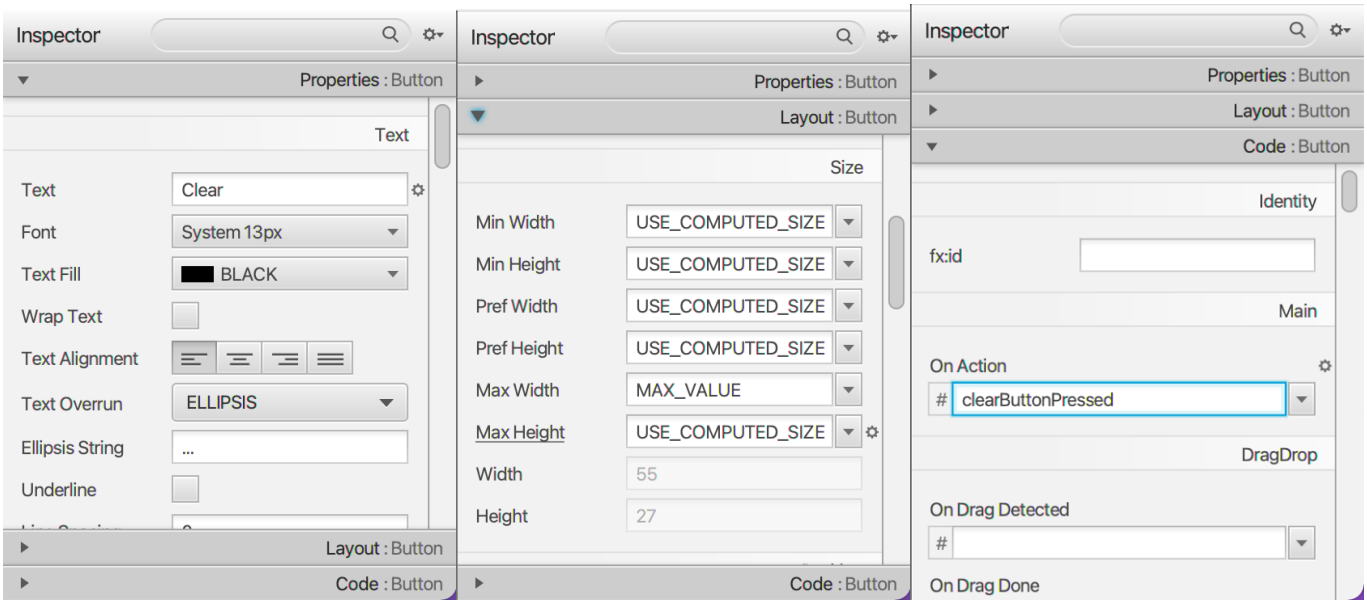

Figure 28. Configuring the Button

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

4.2. Create the controller class

In the same package as the FXML, create a Java class call `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:

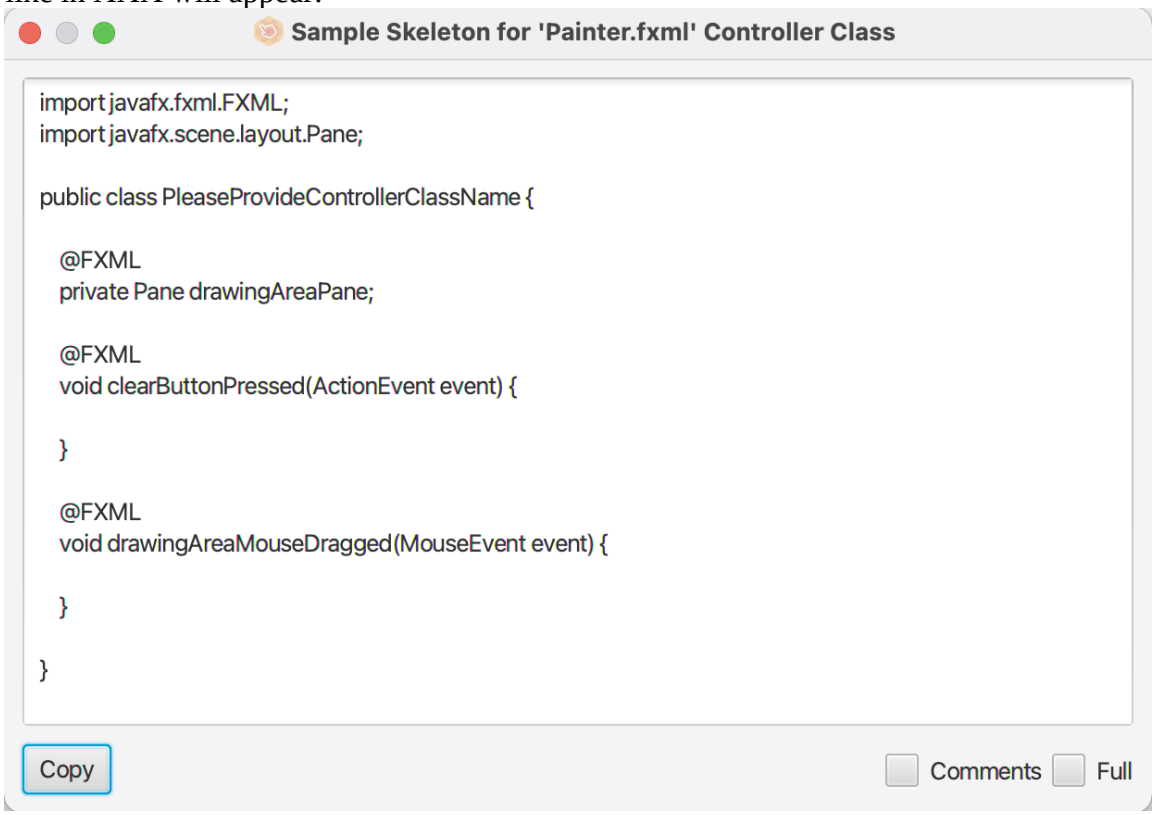

Figure 29. Auto-generated skeleton code for controller

You can choose to copy the skeleton and paste it in 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 30. 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 add the UI object into 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 31.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 32. 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.

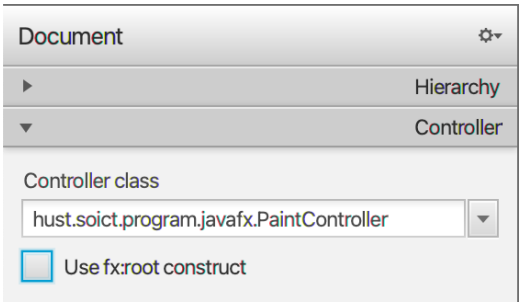

Figure 33. Specify the controller for the FXML file in Scene Builder

4.3. 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 34. Painter source code

Explanation of the code:

- All JavaFX application 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 note. From that root node, a `Scene` is created and set on the `Stage`.

4.4. Practice exercise

Your task now is to add the Eraser feature. The new interface of the app including the new eraser functionality should be like this:

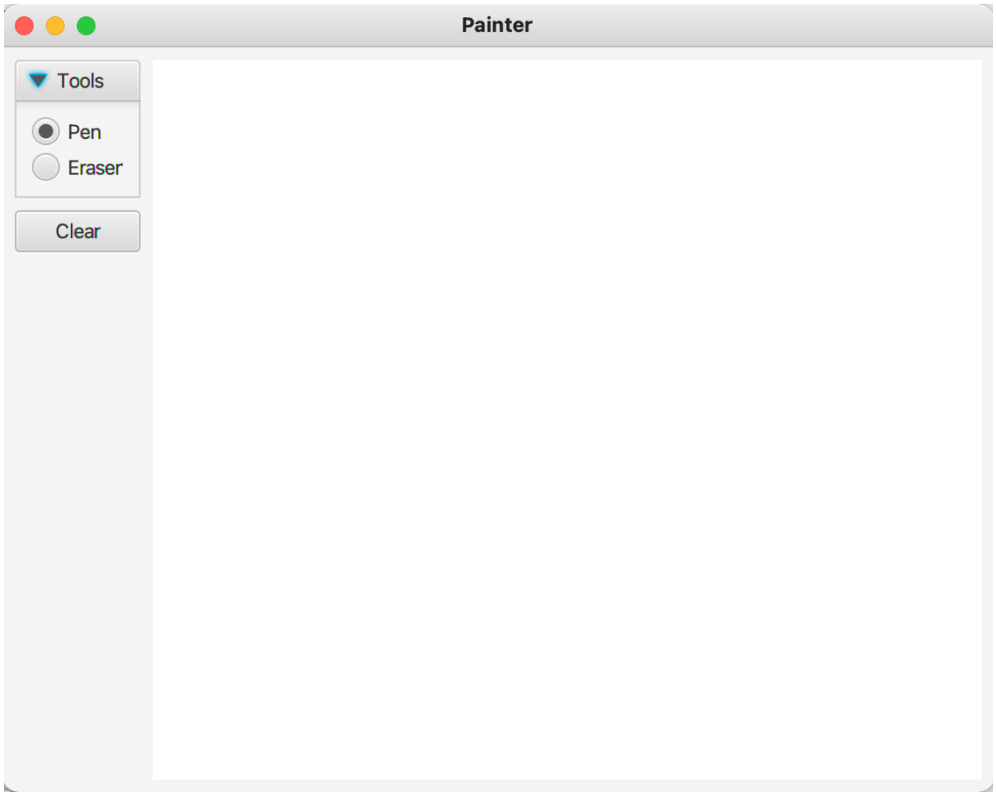

Figure 35. Painter with Eraser

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 a pen above, but use white ink color (canvas color) instead.

5. Setting up the View Cart Screen with ScreenBuilder

Note: For the next 7 exercises for the AimsProject, put the source code under the package "**hust.soict.globalict.aims.screen**" package (for ICT) or the "**hust.soict.dsai.aims.screen**" package (for DS&AI).

We will design a View Cart Screen that looks roughly as below:

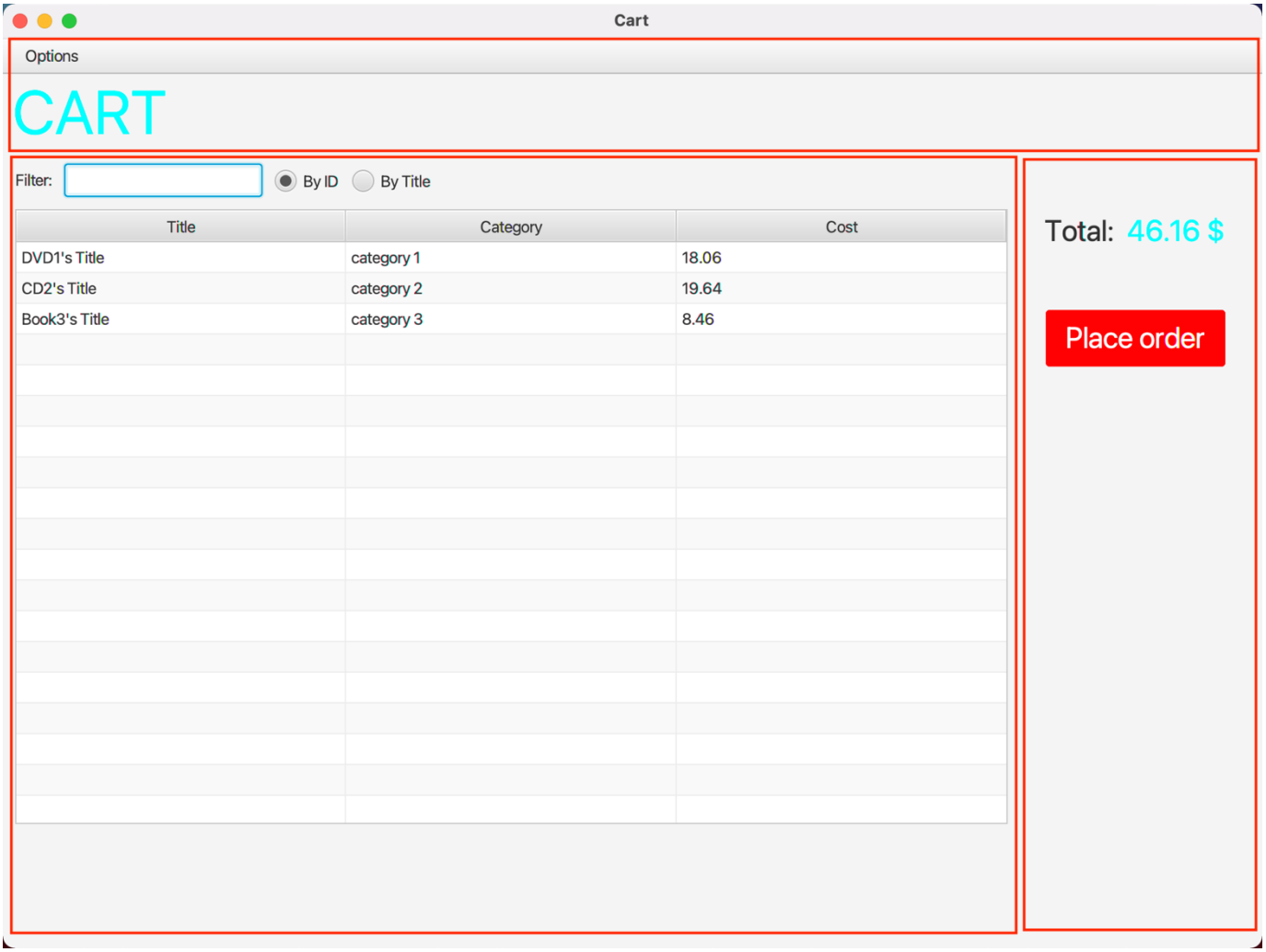

Figure 36. View Cart Screen

Like the previous exercise, we start by creating an FXML file named "**cart.fxml**" with BorderPane being the root node. This layout is the most appropriate for our screen, which clearly has three distinct areas (bounded in red borders) corresponding to TOP, CENTER and RIGHT areas of BorderPane.

5.1. Setting up the BorderPane

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

5.2. Setting up the TOP area

Since the TOP area contains only the MenuBar on top of the header, we will use the Vbox layout.

Step 1. Drag a VBox into the BorderPane's NORTH area.

- Layout: Pref HEIGHT: USE_COMPUTED_SIZE

Step 2. Add a MenuBar into the VBox

- Set up the MenuBar as in Figure 37:

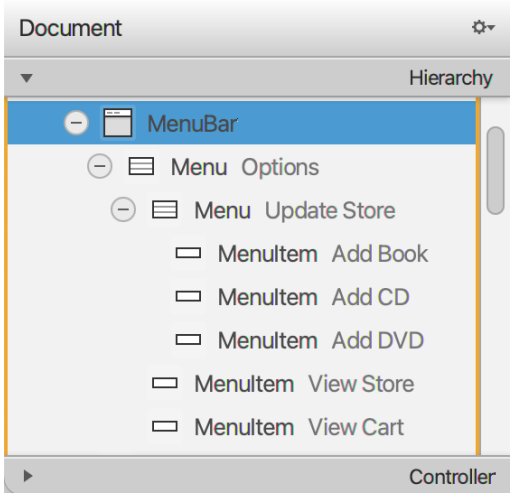

Figure 37. Set up of the MenuBar

Step 3. Add a label into the VBox
- Properties: Text: CART
- Properties: Font: 50px
- Properties: Text Fill: #00ffff (AQUA)
- Layout: Padding: 10 left

5.3. Setting up the CENTER area

Similar to the TOP area, we use the VBox layout for the CENTER areas to arrange components vertically. Inside the Vbox, we use a HBox to arrange the top row of components horizontally. We also use a TableView to display data in a tabular form.

Step 1. Drag a VBox into the CENTER area
- Layout: Padding: 10 left

Step 2. Add a HBox into the VBox
- Properties: Alignment: CENTER_LEFT
- Layout: Padding: 10 top & bottom
- Layout: Spacing: 10
- Layout: Pref Height: USE_COMPUTED_SIZE

Step 2.1. Add a Label into the HBox
- Properties: Text: Filter:

Step 2.2. Add a TextField into the HBox

Step 2.3. Add the first RadioButtons into the HBox
- Properties: Text: By ID
- Properties: Selected: ✓
- Properties: Toggle Group: filterCategory

Step 2.4. Add the second RadioButtons into the HBox
- Properties: Text: By Title
- Properties: Toggle Group: filterCategory

Step 3. Add a TableView into the VBox
- 3 TableColumns: Properties: Text: Title, Category & Cost.
- Column Resize Policy: constrained-resize
- Layout: Pref Width: USE_COMPUTED_SIZE
- Layout: Pref Height: USE_COMPUTED_SIZE

Step 4. Add a ButtonBar into the VBox
- 2 Buttons: Play, Remove

5.4. Setting up the RIGHT area

Step 1. Drag a VBox into the RIGHT area
- Properties: Alignment: TOP_CENTER
- Layout: Pref Width: USE_COMPUTED_SIZE
- Layout: Padding: 50 top

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 into the HBox
- Properties: Text: Total:
- Properties: Font: 24px
- Layout: Spacing: 10

Step 4. Add another Label into the HBox
- Properties: Text: 0 $
- Properties: Font: 24px
- Properties: Text Fill: #00ffff (AQUA)

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

6. Integrating JavaFX into Swing application – The `JFXPanel` class

Previously, we have developed the Aims application using a different GUI library - Swing. Now, we need to find a way to integrate our JavaFX code into this existing Swing application. This exercise is a demonstration on how to do this task.

So far, we have only used the JavaFX Library to create JavaFX applications. In these applications, we use the `load()` method of the `FXMLLoader` class to load an FXML file and create a controller object for it and return a resulting `Node` object. Normally, with a JavaFX application, this `Node` would be put on a `Scene` and the `Scene` will be shown on a `Stage`. However, we can elevate this loading mechanism and embed the `Node` instead in a `JFXPanel`, which can be used in a `JFrame` of a Swing application.

Here is how we will implement this:

```
15 public class CartScreen extends JFrame{
16     private Cart cart;
17
18     public CartScreen(Cart cart) {
19         super();
20
21         this.cart = cart;
22
23         JFXPanel fxPanel = new JFXPanel();
24         this.add(fxPanel);
25
26         this.setTitle("Cart");
27         this.setVisible(true);
28         Platform.runLater(new Runnable() {
29             @Override
30             public void run() {
31                 try {
32                     FXMLLoader loader = new FXMLLoader(getClass()
33                             .getResource("/screen/fxml/cart.fxml"));
34                     CartScreenController controller =
35                             new CartScreenController(cart);
36                     loader.setController(controller);
37                     Parent root = loader.load();
38                     fxPanel.setScene(new Scene(root));
39                 } catch (IOException e) {
40                     e.printStackTrace();
41                 }
42
43
44             }
45         });
46     }
47 }
```

Figure 38. Source code for CartScreen

We create a `CartScreen` class that extends the `JFrame`, just like the `StoreScreen` class. Inside its constructor, we do the following steps:

- In line 23 – 24, we set up a `JFXPanel` in our `JFrame`.
- In line 32 – 37, we load the root `Node` from the FXML file and create its controller object (we will come to the actual implementation of the controller in the next section).
- In line 38, we create a new `Scene` with the root `Node` and add the `Scene` to the `JFXPanel`.

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

The `TableView` we created in the previous exercise is currently empty, we need to fill it with data of media items in our cart. We will do so in our controller class.

In order to manipulate our `TableView` data programmatically in the controller, we need references to the `TableView`'s columns and itself. A convenient way to do this is using Scene Builder. Open the FXML file "**cart.fxml**" earlier and select the TableView element. Set its fx:id property to **tblMedia**. Similary, set the fx:id property for the columns:

- The Title column: **colMediaTitle**
- The Category column: **colMediacategory**
- The Cost column: **colMediaCost**

Create the class `CartScreenController` as follows:

```
12 public class CartScreenController {
13
14     private Cart cart;
15
16     @FXML
17     private TableView<Media> tblMedia;
18
19     @FXML
20     private TableColumn<Media, String> colMediaTitle;
21
22     @FXML
23     private TableColumn<Media, String> colMediacategory;
24
25     @FXML
26     private TableColumn<Media, Float> colMediaCost;
27
28     public CartScreenController(Cart cart) {
29         super();
30         this.cart = cart;
31     }
32
33     @FXML
34     private void initialize() {
35
36         colMediaTitle.setCellValueFactory(
37                 new PropertyValueFactory<Media, String>("title"));
38         colMediacategory.setCellValueFactory(
39                 new PropertyValueFactory<Media, String>("category"));
40         colMediaCost.setCellValueFactory(
41                 new PropertyValueFactory<Media, Float>("cost"));
42         tblMedia.setItems(this.cart.getItemsOrdered());
43
44     }
45 }
```

Figure 39. Source code for CartScreenController

Recall earlier in the `CartScreen` constructor method, the constructor for `CartScreenController` is invoked, then the `load()` method of the FXMLLoader. In the `load()` method, any `@FXML` annotated fields are populated, then `initialize()` is called. That means we use can use the `initialize()` method to perform any post-processing on the content of the `JFrame` after it is loaded from FXML and before it is shown.

In line 42, 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 on 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 40. 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 line 36 – 41, 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 View Cart Screen with some media in your cart, the results will look roughly like in Figure 41

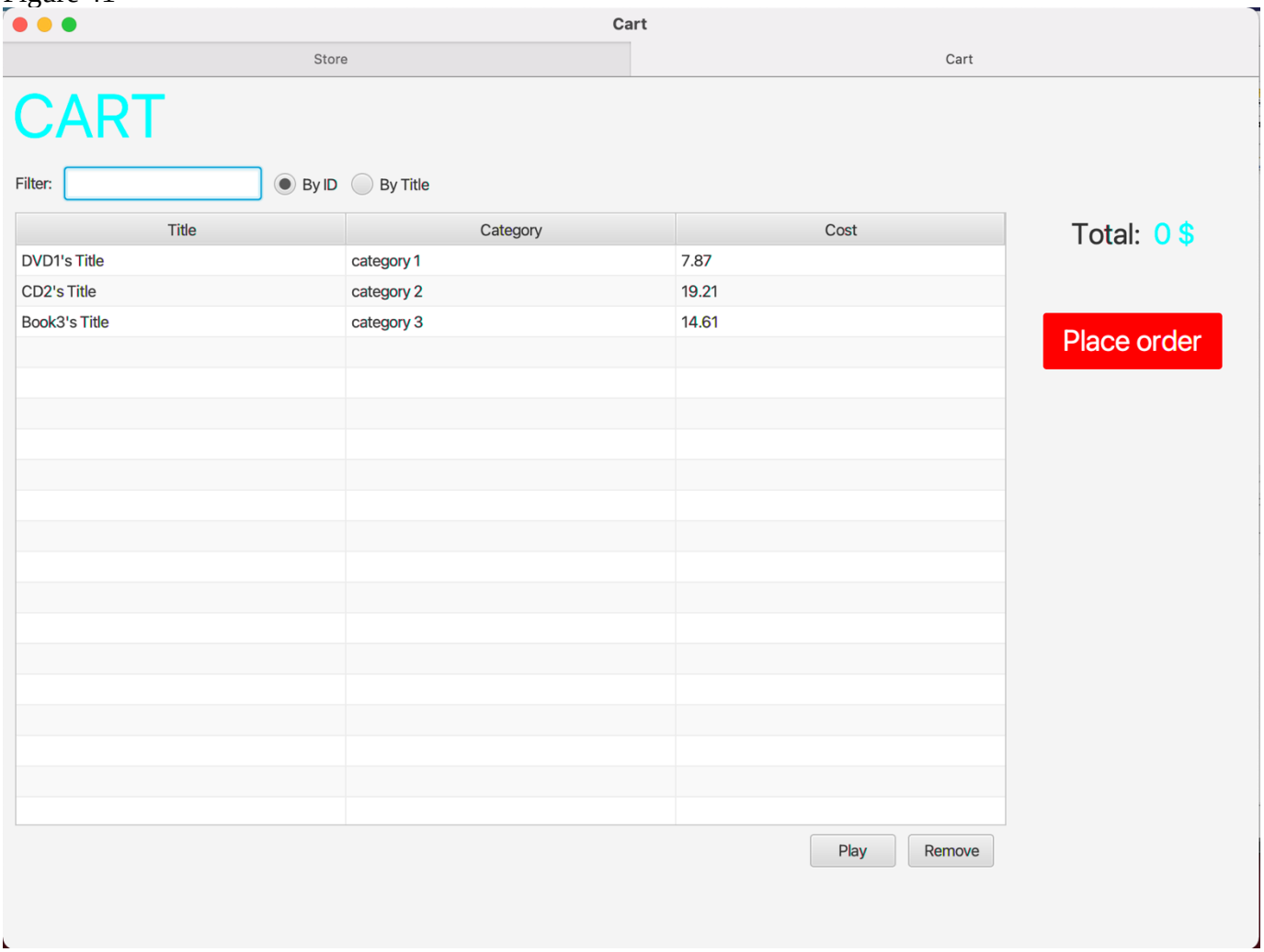

Figure 41. Test displaying TableView data

8. Updating buttons based on selected item in `TableView` – `ChangeListener`

We will now implement a button bar that changes buttons based on the `Media` currently selected in the `TableView`. To do this, again, we need reference to the two buttons in the controller class. Similar to the above, we start by adding the fx:id property for the components in SceneBuilder:

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

Make sure to create the corresponding attributes in `CartScreenController`. You can copy from the Scene Builder's Controller Skeleton with View > Show Sample Controller Skeleton.

First of all, the buttons only appear when a certain `Media` object is being selected, that means we have to modify the `initialize()` method to make them invisible at first (line 73 – 74 in Figure 42)

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

Figure 42. Modified initialize() method

Put some code at the end of the `initialize()` method to add a `ChangeListener` to the `TableView`'s `selectedItem` property (line 76 – 86 in Figure 42). 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 43)

```
114     void updateButtonBar(Media media) {
115         btnRemove.setVisible(true);
116         if(media instanceof Playable) {
117             btnPlay.setVisible(true);
118         }
119         else {
120             btnPlay.setVisible(false);
121         }
122     }
```

Figure 43. Source code of updateButtonBar()

9. 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 as below (change the method name to match the one used by your FXML file):

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

}
```

Figure 44. Handle remove media

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

10. 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 is 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 45):

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

Figure 45. 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.

11. Complete the Aims GUI application

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

- Cart Screen:
  - "Place order" Button
  - "Play" Button
  - The total cost Label - should updated along with changes in the current cart (add/remove).
  - MenuBar
- Store Screen:
  - "Add to cart" Button
- Update Store Screen:
  - When user clicks on one of the items of the "Update Store" menu on the menu bar (such as "Add Book", "Add CD", or "Add DVD"), the application should switch to an appropriate new screen for the user to add the new item. This screen should have the same menu bar as the View Store Screen, so the user can go back to view the store
  - You can choose to use either Swing or JavaFX to implement this
  - For simplicity:
    - you only need to do the "Add item to store" screen, the "Remove item from store" is omitted. As shown above, there is no option of "Remove item from store" in the "Update store" menu.
    - In the "Add item to store" Screen, you don't need to do data type validation yet.
  - Suggestion: You will need to create more screens for this task. Make other modifications to the source code above as needed. You should create 3 classes `AddDigitalVideoDiscToStoreScreen`, `AddCompactDiscToStoreScreen`, and `AddBookToStoreScreen`, which will take the necessary inputs from user. Since

these GUI classes all share part of their interface, if possible, you can apply Inheritance here and create a parent `AddItemToStoreScreen` class and let the above three class inherit from it

- Other features such as: Remove media from store, search media in store, sort media in cart, lucky item, … are optional, you don't have to implement them in the GUI. You can still do them for extra credit.

12. 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 46. Sample exception handling code

13. 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`**.

*13.1. 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

*13.2. 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 47. Sample code for method play() of DigitalVideoDisc

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

*13.3. Update **play()** in the **Playable** interface*

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

*13.4. 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 48. 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.

14. 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.

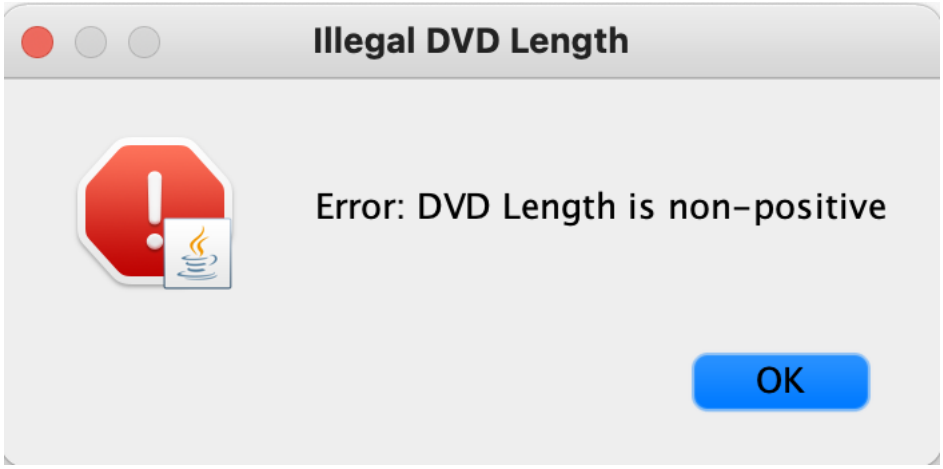

Figure 49.Swing dialog showing error

15. Modify the `equals()` method of `Media` class

- Two medias are equal if they have the same `title`

- 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`.

16. 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

17. Update Aims class diagram

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