Object-Oriented Programming

Lecturer: NGUYEN Thi Thu Trang, trangntt@soict.hust.edu.vn

Lab 04: GUI Programing - Part 1

In this lab, you will practice with:

- Re-organize the application with a menu-based UI for users.

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 three days after the class**: for this deadline, you should include the **source code** of all sections of this lab, into a branch namely "**release/lab04**" 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 own work and not give or receive unpermitted aid. Otherwise, we would apply extreme methods for measurement to prevent cheating. Please note that you need to write down answers for all questions into a text file named "**answers.txt"** and submit within it within your repository.

1. Swing components

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.dsai.swing"** for DSAI.

Note: From this section onwards, it is assumed that you are a DS-AI student, so your folder structure will contain the **"dsai"** package. If you are an HEDSPI or ICT student, you should replace the **"dsai"** string with **"hedspi"** or **"globalict"**.

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.

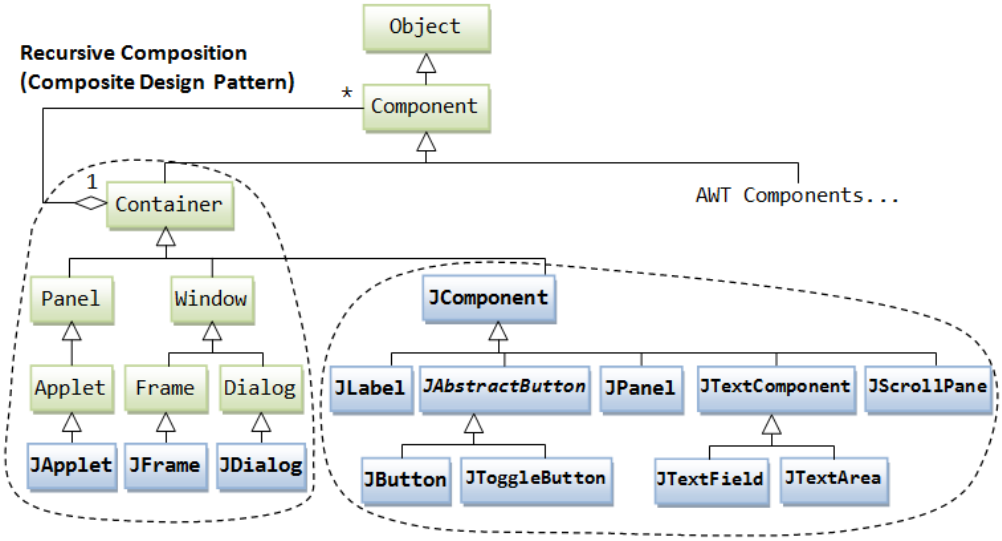

Figure 4. AWT and Swing elements

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 a 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`

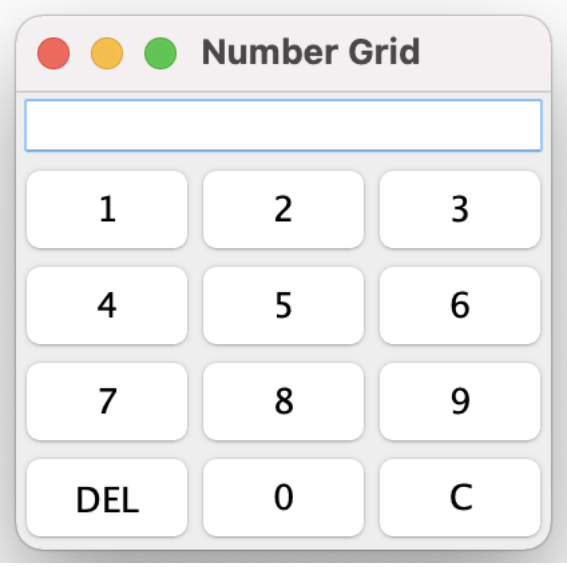

Figure 5. NumberGrid

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 and put all your source code in the package **"hust.soict.dsai.aims.screen"** for DSAI.

3.1. View Store Screen

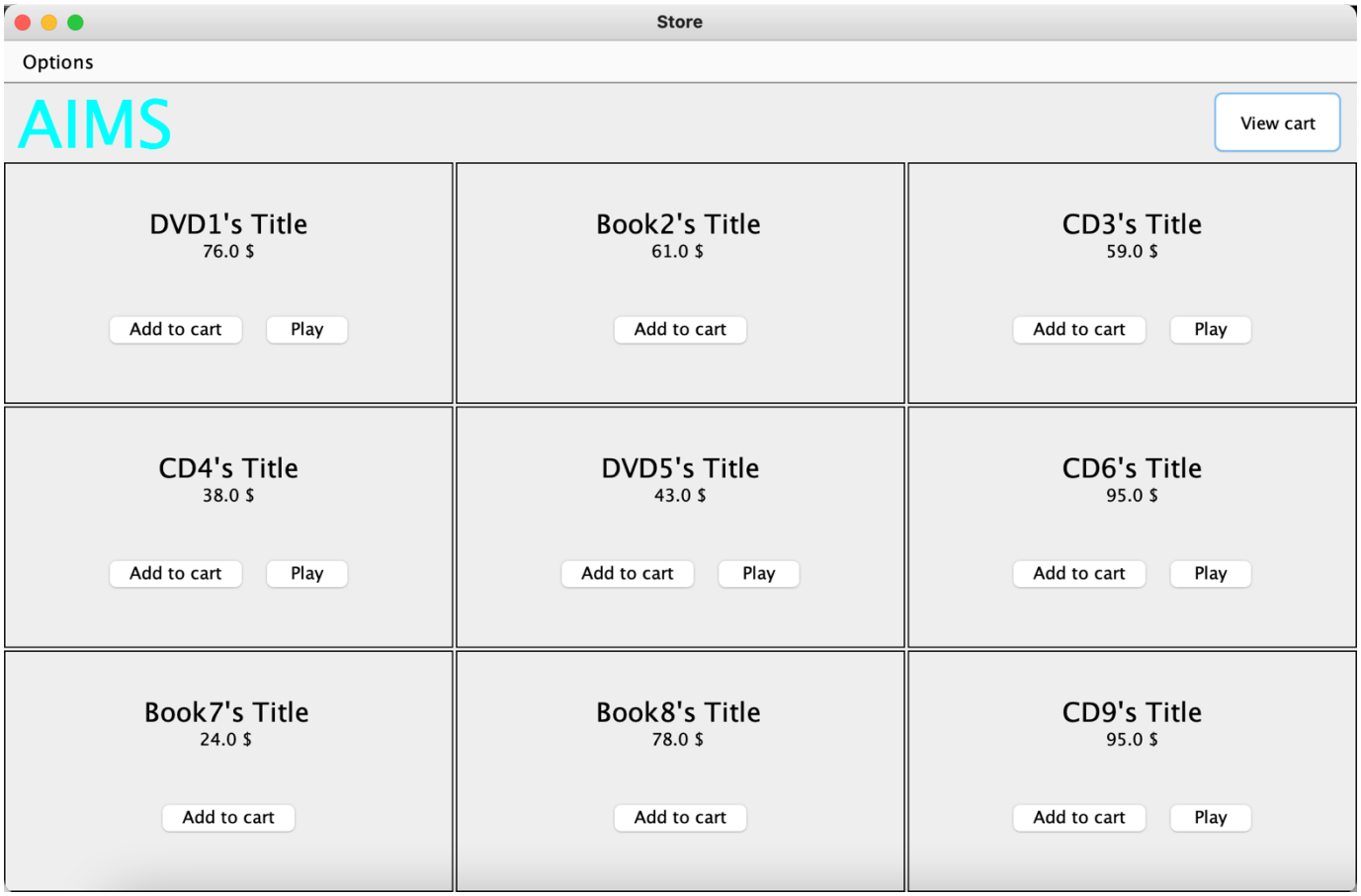

Figure 9. View Store Screen

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:

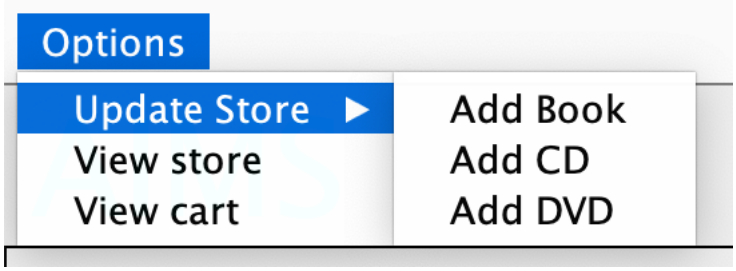

Figure 13. Resulting menu on menu bar

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 the user clicks on the "Play" button, the Media should be played in a dialog window. You can use `JDialog` here.
  - When the 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.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 which allows users to draw on a white canvas with their mouse.

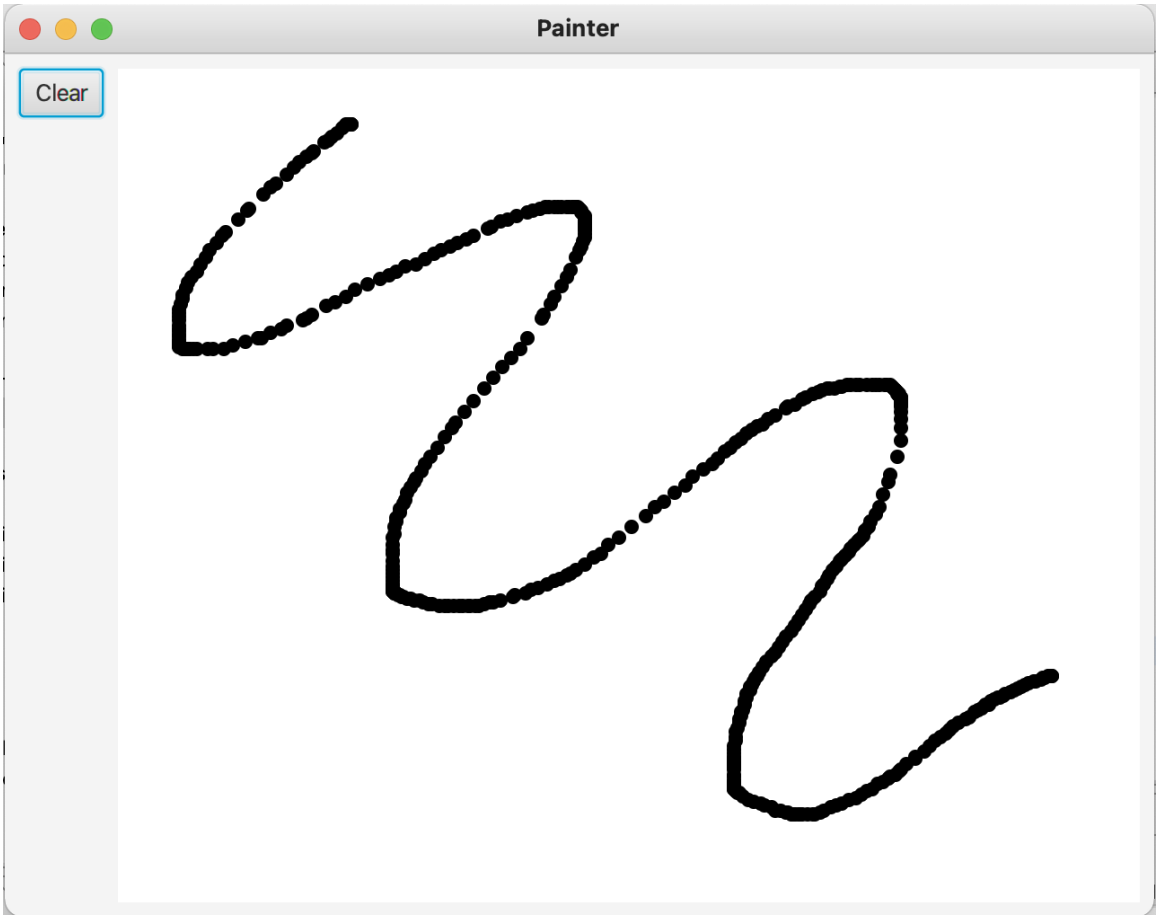

Figure 18. Painter app

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`).

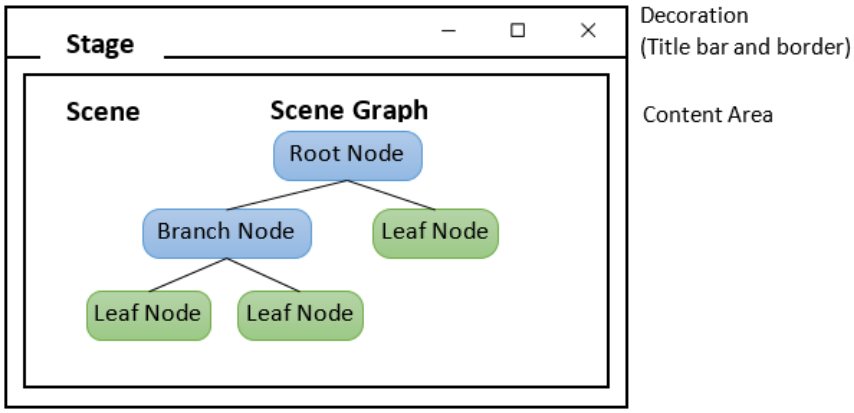

Figure 19. Structure of JavaFX application

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 adds 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.

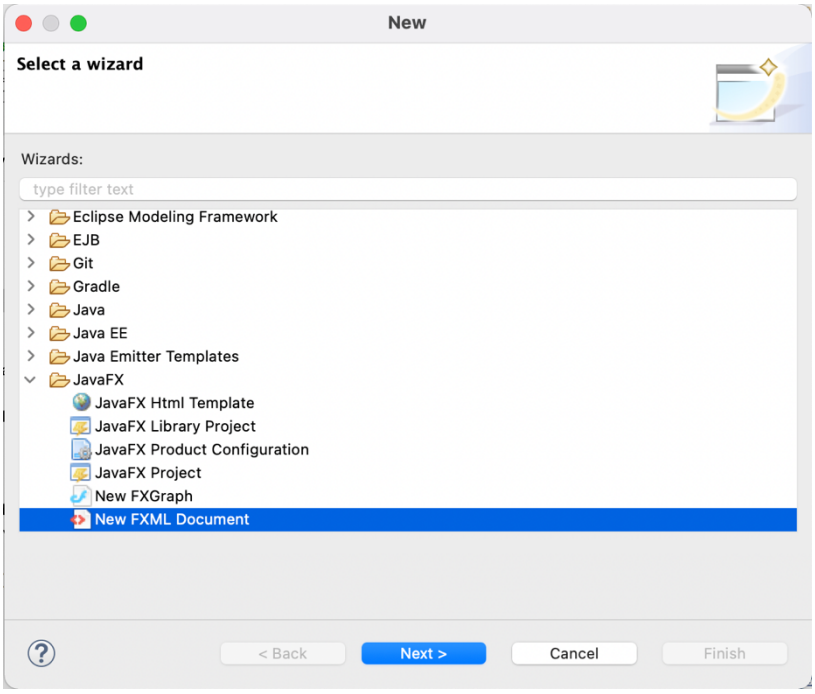

Figure 20. Create a new FXML in Eclipse(1)

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

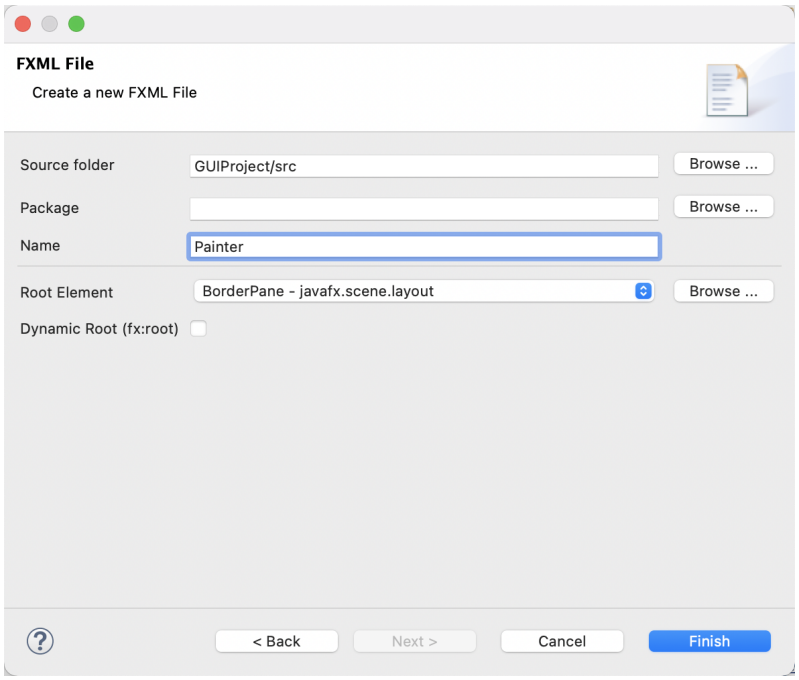

Figure 21. Create a new FXML in Eclipse(2)

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

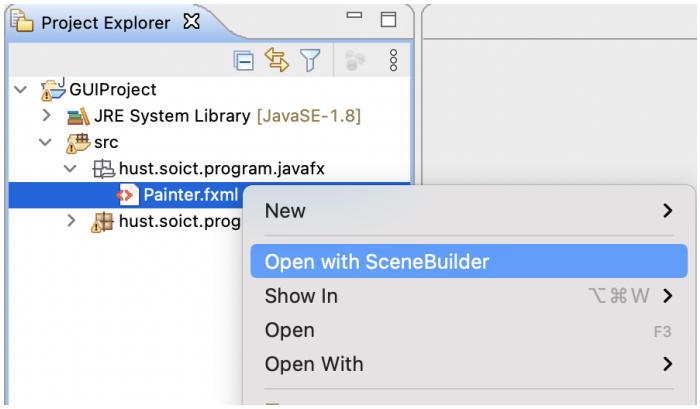

Figure 22. Open FXML with SceneBuilder from Eclipse

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.

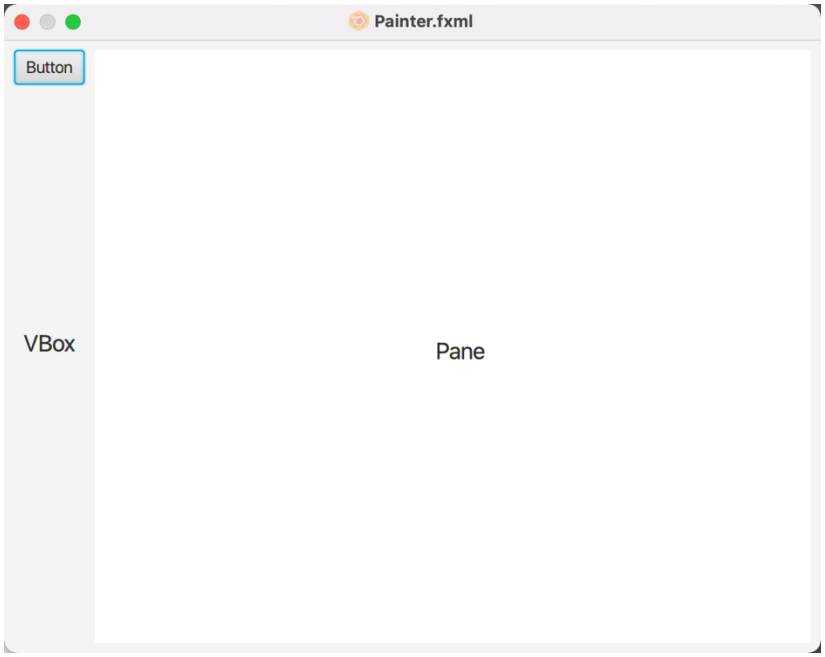

Figure 23. Target interface

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 insert it from the stage's edges

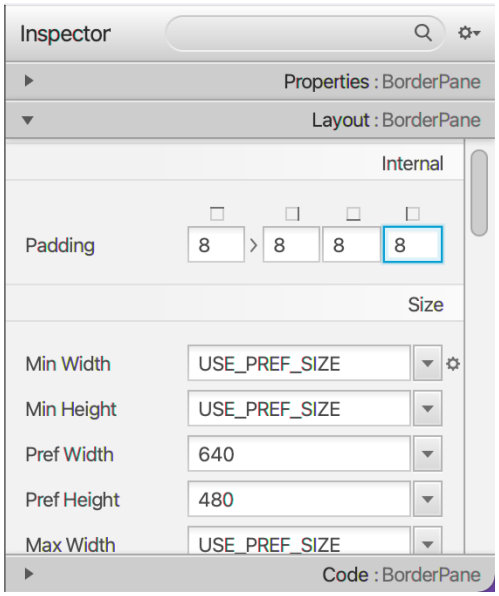

Figure 24. Configuring the BorderPane

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.

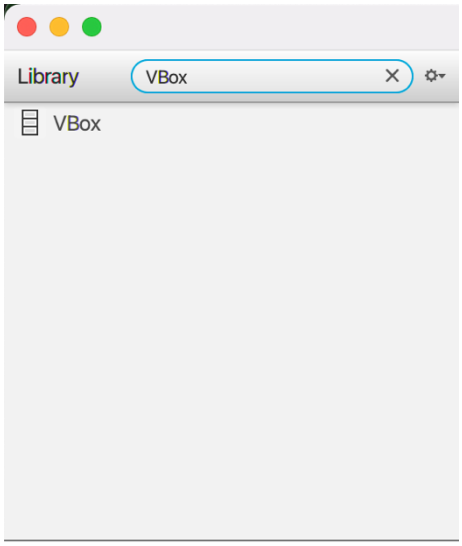

Figure 25. Get VBox in the Library menu

- 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)

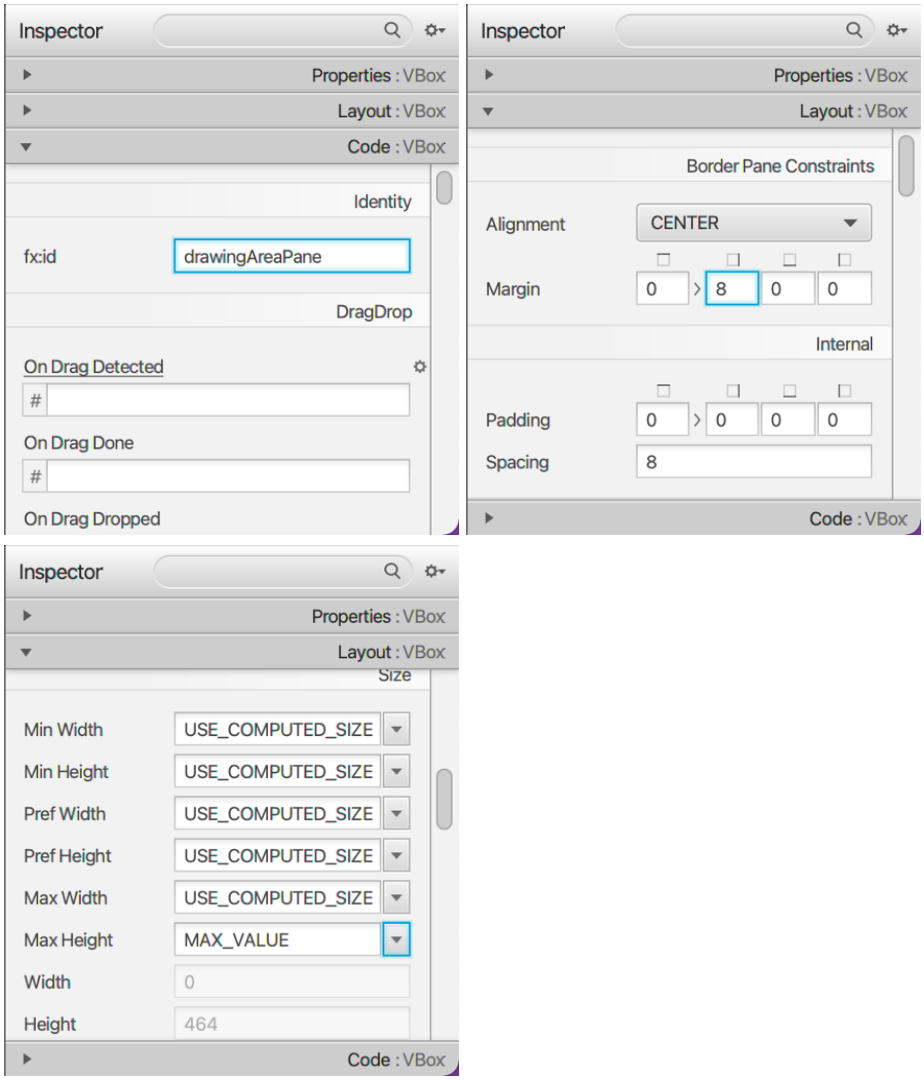

Figure 26. Configuring the VBox

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.

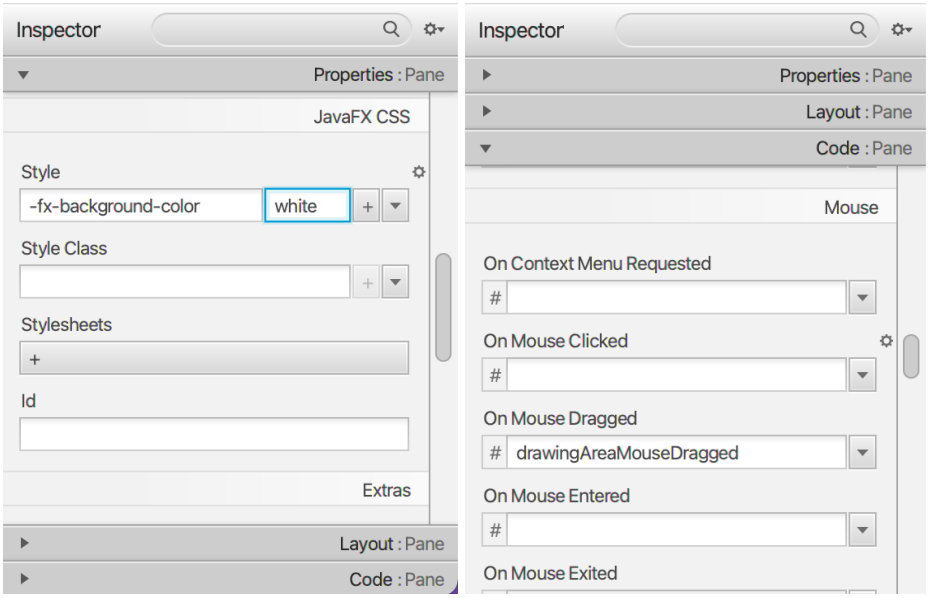

Figure 27. Configuring the Pane

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.

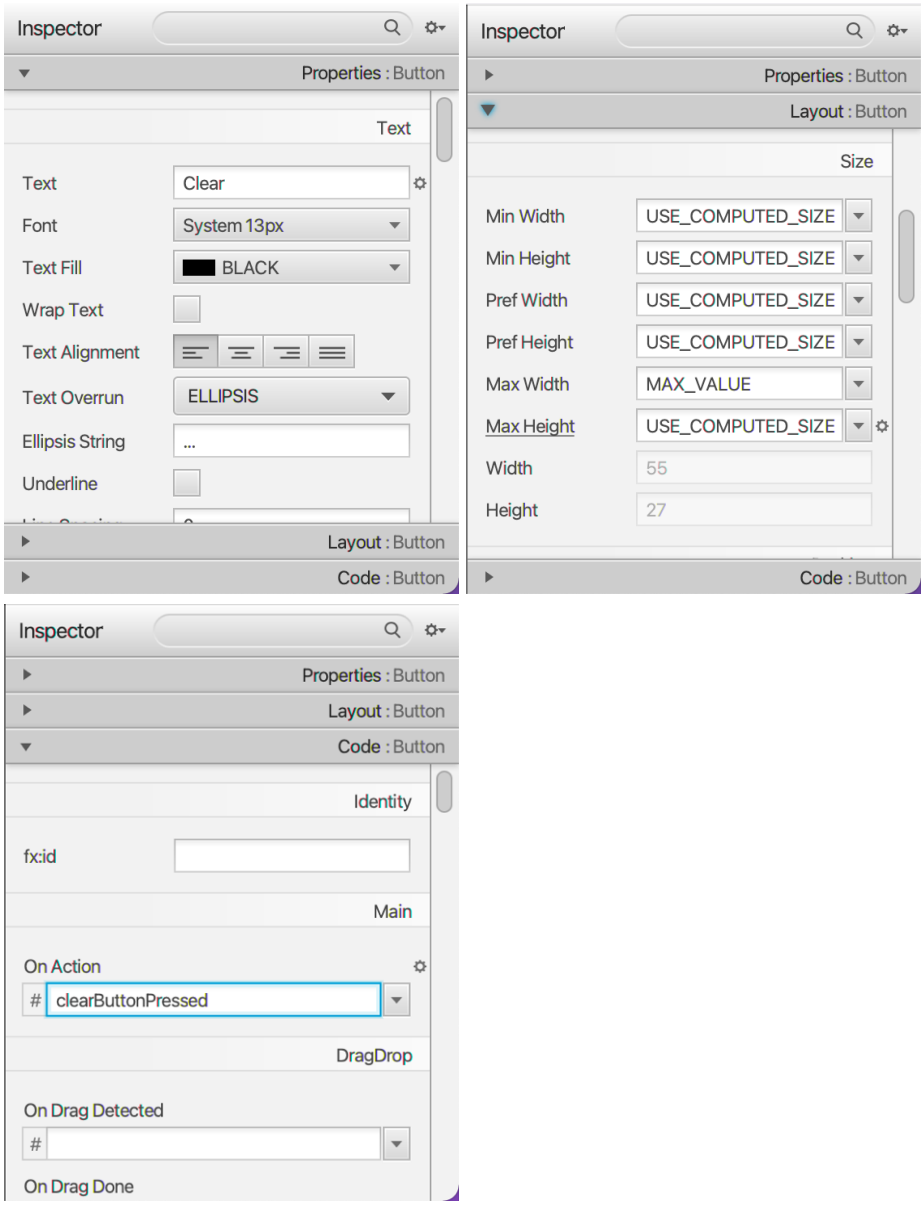

Figure 28. Configuring the Button

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 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:

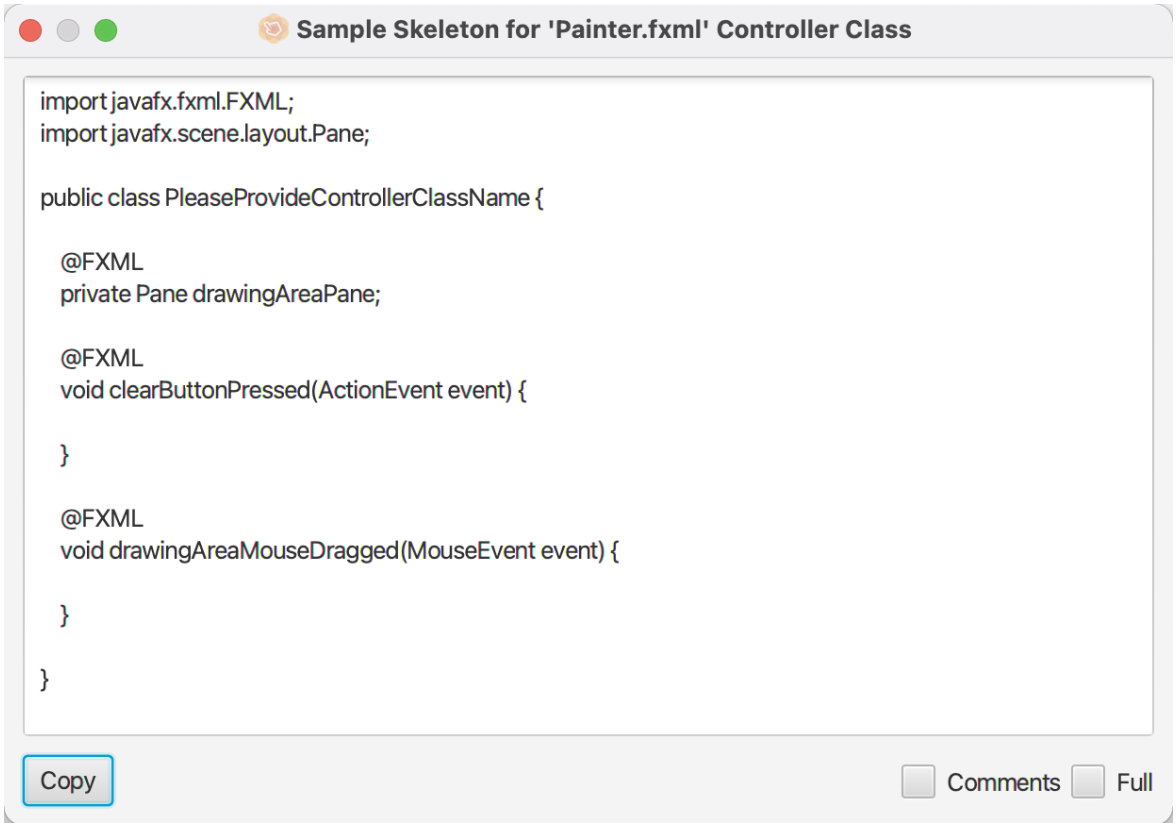

Figure 29. Auto-generated skeleton code for controller

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.

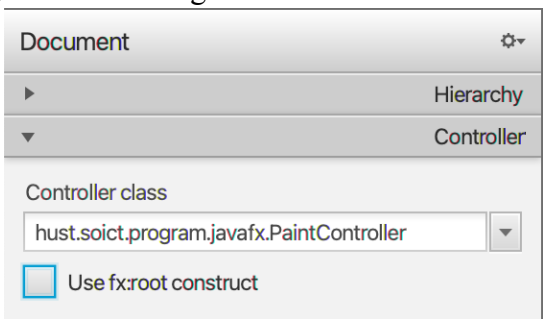

Figure 33. Specify the controller for the FXML file in Scene Builder

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 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 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:

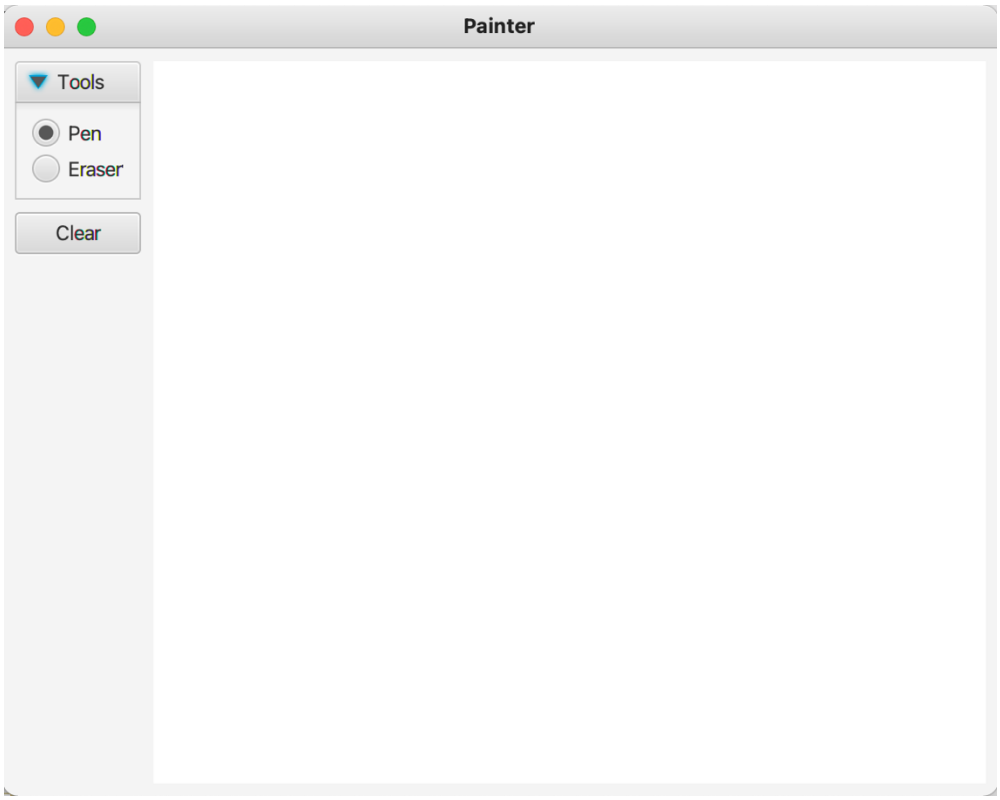

Figure 35. Painter with Eraser

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.