

Name: _____ () Date: _____

Chapter 14: JavaFX With Scene Builder

Lesson Objectives

After completing the lesson, the student will be able to

- Creating JavaFX program using Scene Builder
- layout nodes using Pane, StackPane, FlowPane, GridPane, BorderPane, HBox, and VBox
- understand and write code for event handling
- use advanced features such as Alert Dialog Boxes, Controls such as CheckBoxes etc

14.1 Overview of MVC Design Pattern

14.1.1 What is MVC?

MVC (Model-View-Controller) is a pattern in software design commonly used to implement user interfaces(view), data (model), and controlling logic (controller). It emphasizes a separation between the software's business logic and display. This "separation of concerns" provides for a better division of labor and improved maintenance. Some other design patterns are based on MVC, such as MVVM (Model-View-Viewmodel), MVP (Model-View-Presenter), and MVW (Model-View-Whatever).

The three parts of the MVC software-design pattern can be described as follows:

- Model: Manages data and business logic.
- View: Handles layout and display.
- Controller: Routes commands to the model and view parts.

We will describe how we could implement a shopping cart app using MVC.

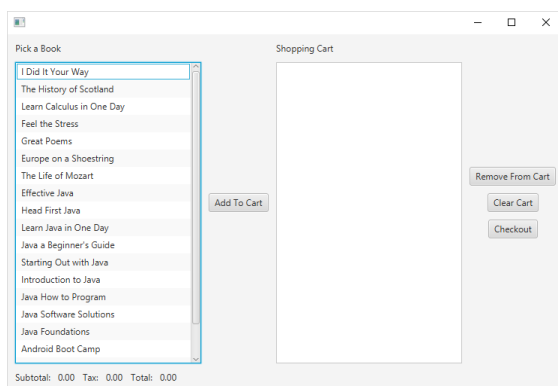


Figure 1: Shopping Cart UI

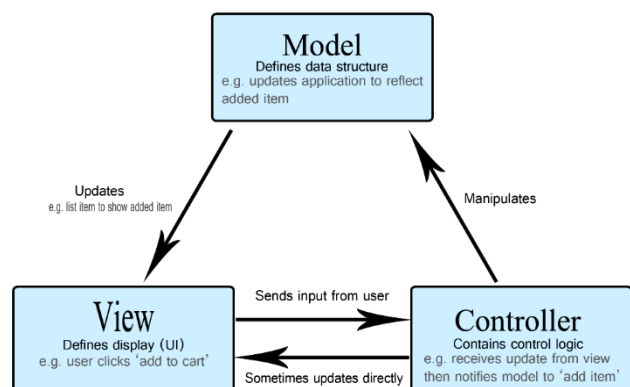


Figure 2: MVC

The Model

The model defines what data the app should contain. If the state of this data changes, then the model will usually notify the view (so the display can change as needed) and sometimes the controller (if different logic is needed to control the updated view).

Going back to our shopping app, the model would specify what data the list items should contain — item, price, etc. — and what list items are already present.

There could be a class named *Book* to store information of the Books such as title, author, price etc. A class named *Database* to store an *ArrayList* of various books in store, and books in cart.

The View

The view defines how the app's data should be displayed.

In our shopping app, the view would define how the list is presented to the user, and receive the data to display from the model.

In the UI presented in Figure 1, the title of books available are listed in a *ListView* control. There are buttons to allow user to add and delete products to shopping cart etc.

The Controller

The controller contains logic that updates the model and/or view in response to input from the users of the app.

So for example, our shopping app contains buttons that allow us to add or delete books. These actions require the model (e.g. *ArrayList* of books) to be updated, so the input is sent to the controller, which then manipulates the model as appropriate, which then sends updated data to the view in order to present the updated shopping cart.

You might however also want to just update the view to display the data in a different format, e.g., change the item order to alphabetical, or lowest to highest price. In this case the controller could handle this directly without needing to update the model.

14.1.2 MVC in Pure Code Style

Recall in Chapter 13, we implemented our JavaFX program using pure Java code. How then will MVC be implemented using the Java code? Let's take a look at the HelloWorld example:

```
//Application is the main class of a JavaFX program. Each JavaFX program must extend the
//Application class. This means that each FX program is a "child" of Application class.
//Thus all JavaFX program will inherit the attributes and methods from the Application class
public class HelloWorldFXCode extends Application {
    private Label label; //create a variable to reference a Label object
    private Button btn; //create a variable to reference a Button object

    @Override //start() method is the main entry point of the application
    public void start(Stage primaryStage) {

        btn = new Button("Click Me"); //creates a button object
        label = new Label(); //creates a label object

        //connect btn to the code that runs when you click it
        btn.setOnAction(e -> buttonClick(e));

        FlowPane root = new FlowPane(); //creates a pane to contain UI
        root.getChildren().add(btn); //add the button to the pane
        root.getChildren().add(label); //add the label to the pane

        //create the scene object with specific dimensions. Add the pane into the scene
        Scene scene = new Scene(root, 300, 250);

        primaryStage.setScene(scene); //add the scene to the stage
        primaryStage.setTitle("Hello World!"); //set title of stage (i.e. window title bar)
        primaryStage.show(); //show the stage (window)
    }

    //code to be triggered when user clicks on the button
    public void buttonClick(ActionEvent event) {
        System.out.println("You clicked me!");
        label.setText("Hello World!"); //change text of label. Note that you are able
        //to access label in this method as it's instance var
    }

    //main method is not necessary for JavaFX program.
    //It is only used as a fallback for situations in which JavaFX launching is not working.
    public static void main(String[] args) {
        launch(args);
    }
}
```

View:

Code that setups the UI will be the View.

Controller:

Code that does event handling is the controller

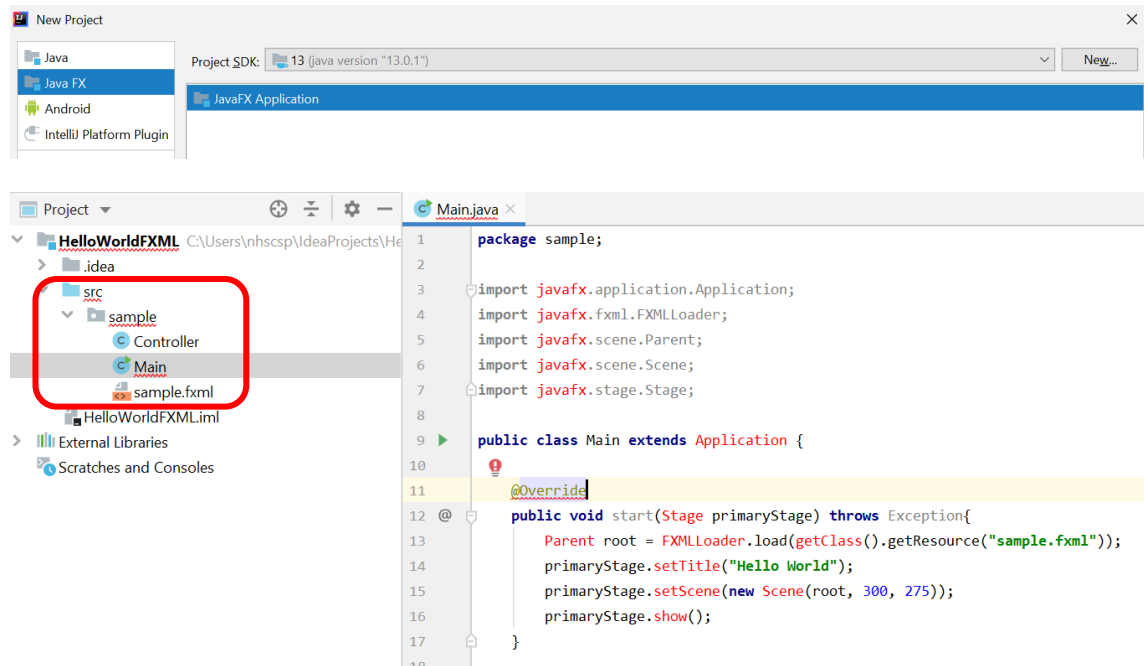
Model:

In this example, there is no model.

In a larger application, separate classes may be created for View, Controller and Model.

14.1.3 MVC in JavaFX Project created via IntelliJ

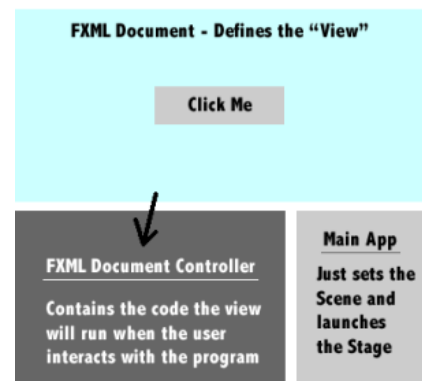
In general, when you create a JavaFX Project in IntelliJ, three files will be created for you as shown below:



The diagram on the right aptly summarizes the purpose of the 3 files (Main.java, sample.fxml and Controller.java).

From a **Model View Controller** (MVC) perspective,

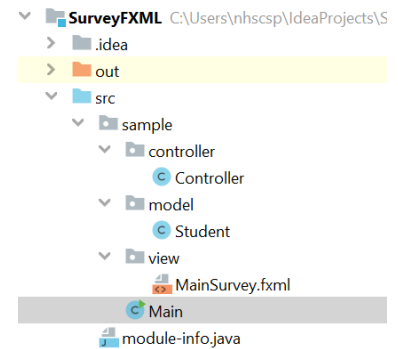
- the FXML file that contains the description of the user interface (in XML code) is the **view**.
- The **controller** is a Java class (Controller.java) to control the logic of the program. It will contain code to handle events (i.e. what should happen when user does something via the UI e.g. click on a button).
- The **model** consists of domain objects, defined on the Java side (e.g. if you are coding a shopping cart application, you may have Java objects such as Book), that you connect to the view through the controller (not implemented in this project).
- Finally, Main.java will usually just contain code to load the view and launch the program.



```
public class Main extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception{
        Parent root = FXMLLoader.load(getClass().getResource("sample.fxml"));
        primaryStage.setTitle("Hello World");
        primaryStage.setScene(new Scene(root, 300, 275));
        primaryStage.show();
    }
}
```

For larger scale JavaFX program, to better organize your code, it is also a common practice to **package the classes related to View, Model and Controller into 3 different packages.**

The figure on the right shows an example of how packaging can be applied. In large scale JavaFX program, you are likely to have multiple controller classes, model classes and different FXML files for different views (e.g. you may have more than 1 stage).



14.2 Understanding the Scene Builder

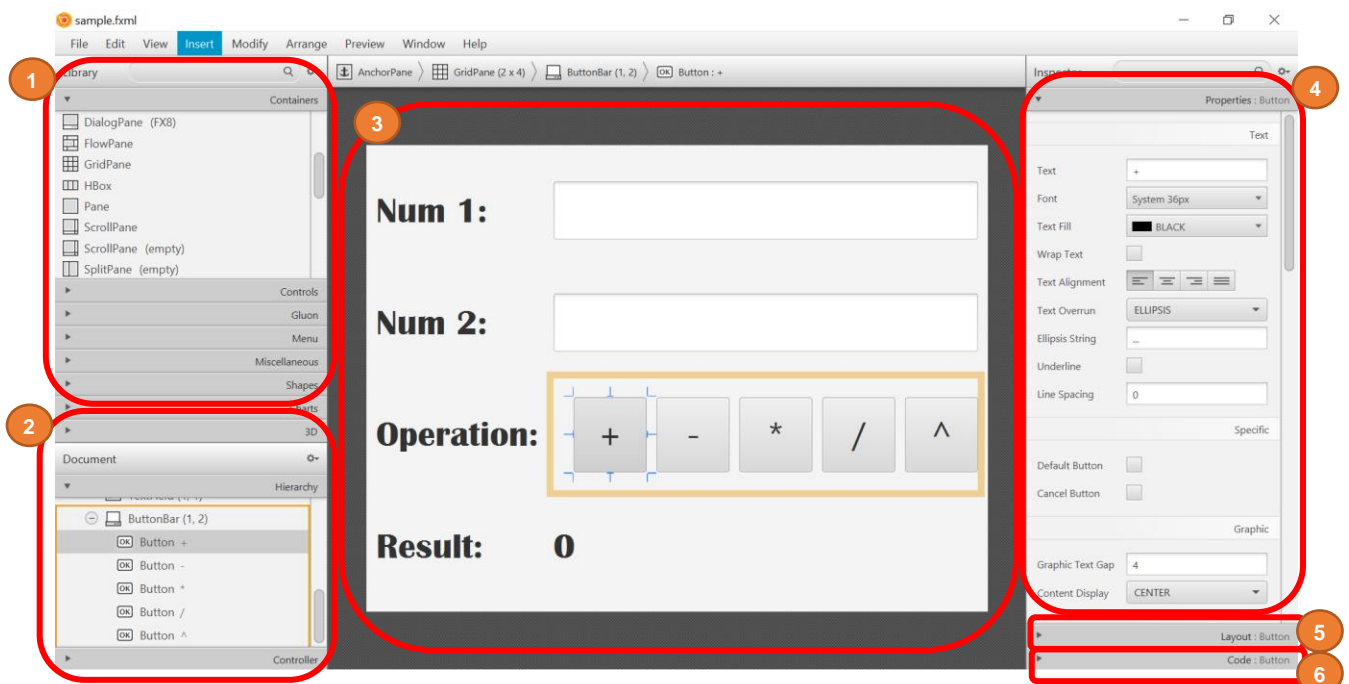
As shown in the sample JavaFX project in above, the view of a JavaFX project created in IntelliJ is stored in a file named sample.fxml. You may layout your view (GUI) in sample.fxml using the Scene Builder application.

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.

Scene Builder can help you to quickly create a prototype for an interactive application that connects components to the application logic.

To use Scene Builder, you will first need to install Scene Builder at <http://gluonhq.com/open-source/scene-builder/>.

In this chapter, we will go into details on how to create a JavaFX program with the help of Scene Builder. But before we do that, let's try to understand the different sections in Scene Builder.



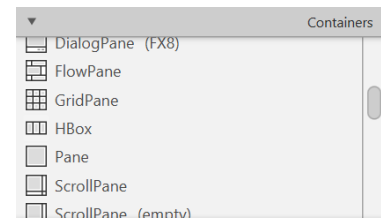
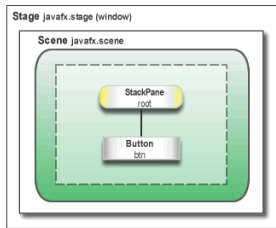
We will explain the 6 sections in Scene Builder in greater detail below.

14.2.1 Section 1 – Library

The Library contains the UI elements for you to add to your graphical user interface.

Containers


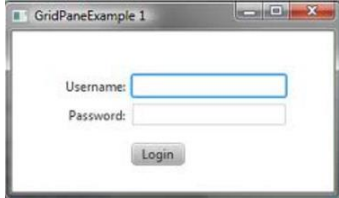

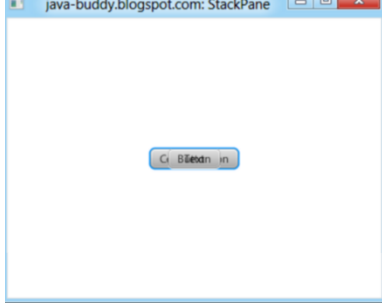

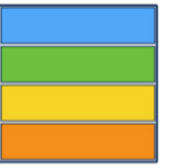
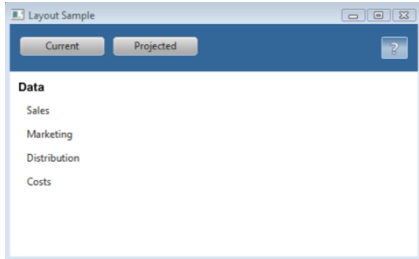
Containers consist of the layout managers (e.g. FlowPane, GridPane etc).



Recap that a scene graph is contained in a scene. The first node (or root node) in the scene graph is normally a Pane object. A Pane is really a layout manager.

JavaFX provides several layouts out of the box, which can be seen in the following diagram:

<p>AnchorPane</p>		<p>The AnchorPane layout pane enables you to anchor nodes to the top, bottom, left side, right side, or center of the pane. As the window is resized, the nodes maintain their position relative to their anchor point. Typically we will start with an AnchorPane, and add other layout manager into the AnchorPane.</p>
<p>BorderPane</p>		<p>The BorderPane layout pane provides five regions in which to place nodes: top, bottom, left, right, and center. This layout is useful when you want to have a menubar at the top and footer notes at the bottom.</p>
<p>FlowPane</p>		<p>The nodes within a FlowPane layout pane are laid out consecutively and wrap at the boundary set for the pane.</p>
<p>TilePane</p>		<p>A tile pane is similar to a flow pane. The TilePane layout pane places all of the nodes in a grid in which each cell, or tile, is the same size</p>

 <p>GridPane</p>		<p>The GridPane layout pane enables you to create a flexible grid of rows and columns in which to lay out nodes. Nodes can be placed in any cell in the grid and can span cells as needed. A grid pane is useful for creating forms or any layout that is organized in rows and columns.</p>
 <p>StackPane</p>		<p>The StackPane layout pane places all of the nodes within a single stack with each new node added on top of the previous node. This layout model provides an easy way to overlay text on a shape or image or to overlap common shapes to create a complex shape.</p>
 <p>HBox</p>  <p>VBox</p>		<p>The HBox layout pane provides an easy way for arranging a series of nodes in a single row.</p> <p>The VBox layout pane is similar to the HBox layout pane, except that the nodes are arranged in a single column.</p>

For more information, proceed to http://docs.oracle.com/javafx/2/layout/builtin_layouts.htm

Other useful pane include SplitPane (which splits the frame into two windows), Tab Pane etc. You may explore them yourself via Scene Builder.

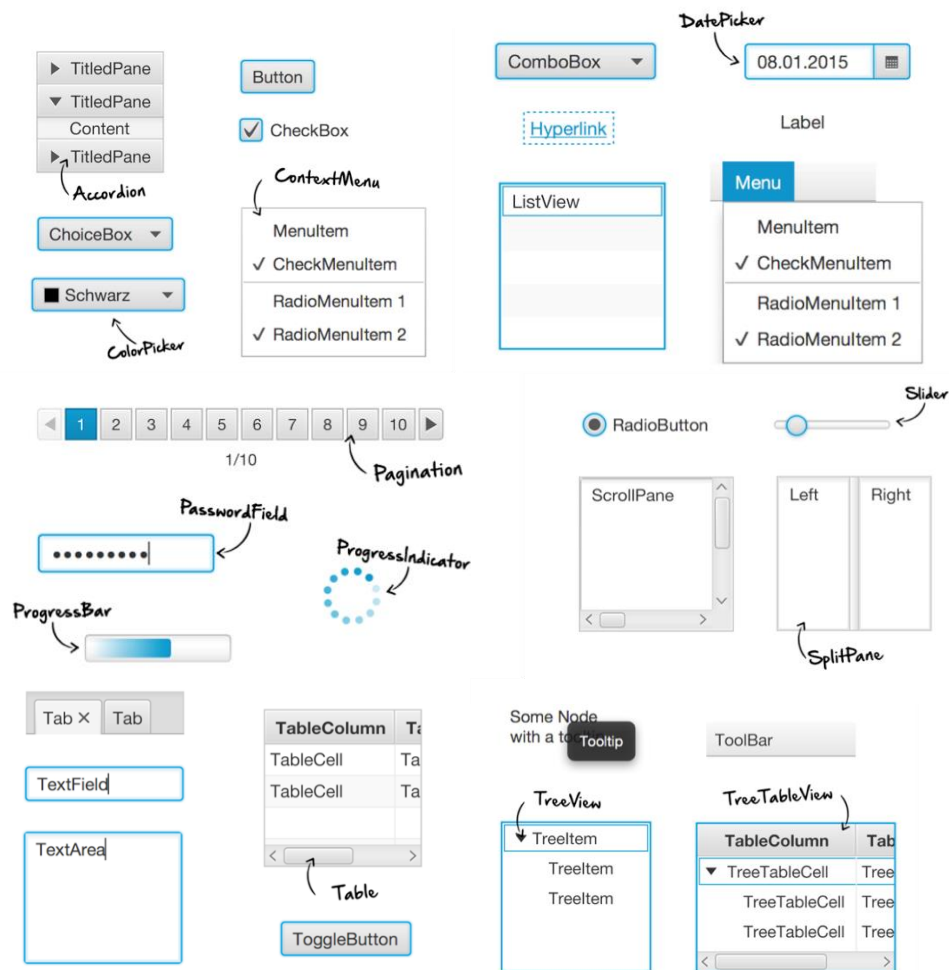
Finally, note that to create rich layout, it is common to nest a layout manager inside another layout manager (e.g putting a Flow Pane in one cell of a GridPane etc). Generally, BorderPane and AnchorPane are the two more popular choices for the first pane in your main window. You can then nest other panes in them.

Controls

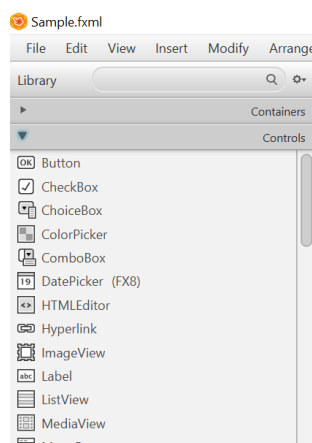
Layout manager is the “bottom” layer in our scene. Once we set the layout, we can start adding controls into the layout manager.

Controls are the JavaFX UI nodes that a developer will use most of the time. All controls—like buttons, textfields, or tables—extend the Control class.

Here’s a quick overview of some controls:



In Scene Builder, the common controls are found under the “Controls” section as shown below:

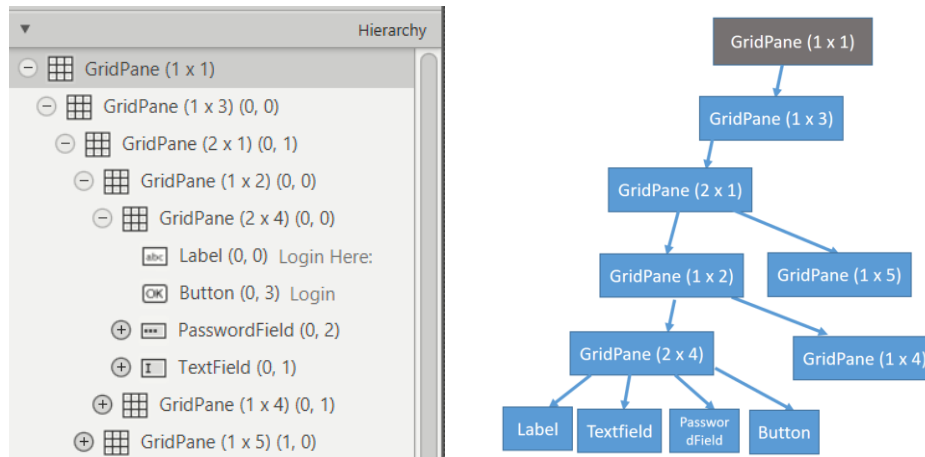


14.2.2 Section 2 – Document

Hierarchy

Hierarchy shows all the UI elements that compose your FXML layout. You can use the Hierarchy panel to focus on one specific UI element, whether it is a parent node or a leaf node.

For example, the following Hierarchy will translate into the figure on the right in a scene graph.

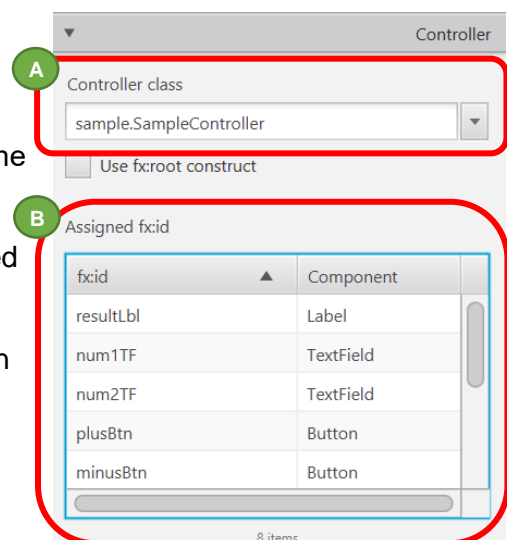


You should use the hierarchy to help plan out your UI. Designing a good and fully functioning functional UI takes A LOT of planning and consideration!

Controller

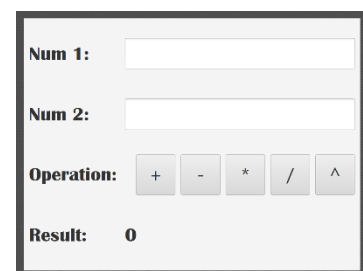
The controller panel consist of 2 important section.

- Controller class allows you to “link” the view (i.e. the FXML file) with the Controller class (i.e. Controller.java)
In this example, the view is linked to a class named SampleController, found in package sample.
- Assigned fx:id lists the UIs that has been named in the View. It shows a quick summary of all the controls you have added to your UI, and their respective variable names.



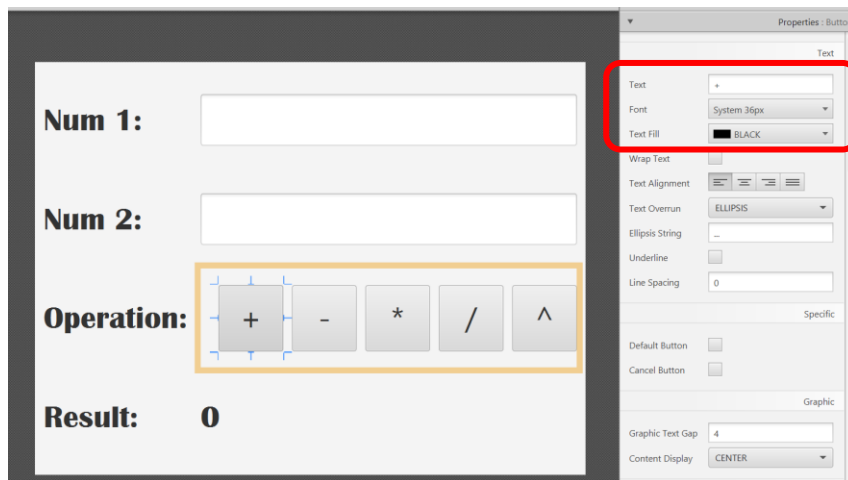
14.2.3 Section 3 – The GUI

The middle panel shows how your GUI will look like when the program is executed.



14.2.4 Section 4 – Properties

Properties panel allows you to define the look and feel of the UI currently selected. For example, in the figure below, + button is currently selected. Changing the Text Fill, Font or Font Size in the Properties panel will result in changes to the + button's text color, font and size.



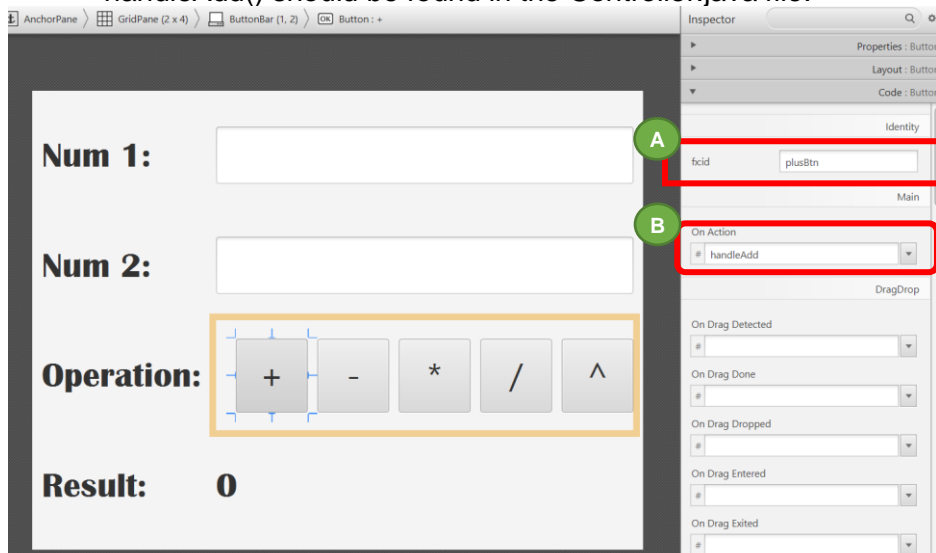
14.2.5 Section 5 – Layout

The Layout section helps you to specify the runtime behavior of the layout when the application's window is resized. It also enables you to change the size (such as, Pref Width and Pref Height) and position (such as, Layout X and Layout Y) of the selected element.

14.2.6 Section 6 – Code

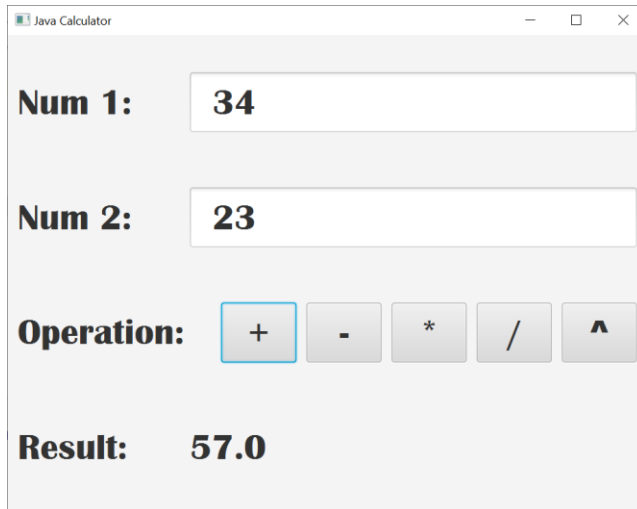
The Code panel allows you to define a few important things to tie the FXML to your Controller.java class.

- A) fx:id – you define the name of the control in the fx:id textbox. For example, the variable name of the + Button is plusBtn as shown below.
- B) On Action – you define the name of the eventhandler in this textbox. For example, when + Button is clicked, the eventhandler named handleAdd will be triggered. The code for handleAdd() should be found in the Controller.java file.



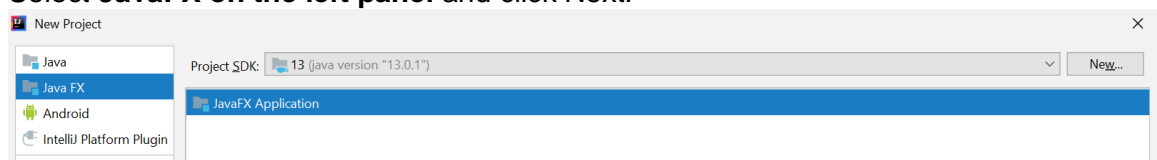
14.3 Creating a JavaFX Program using Scene Builder

Let's go through how to create a JavaFX program using Scene Builder. In this section, we will build a simple JavaFX Calculator App as shown below:

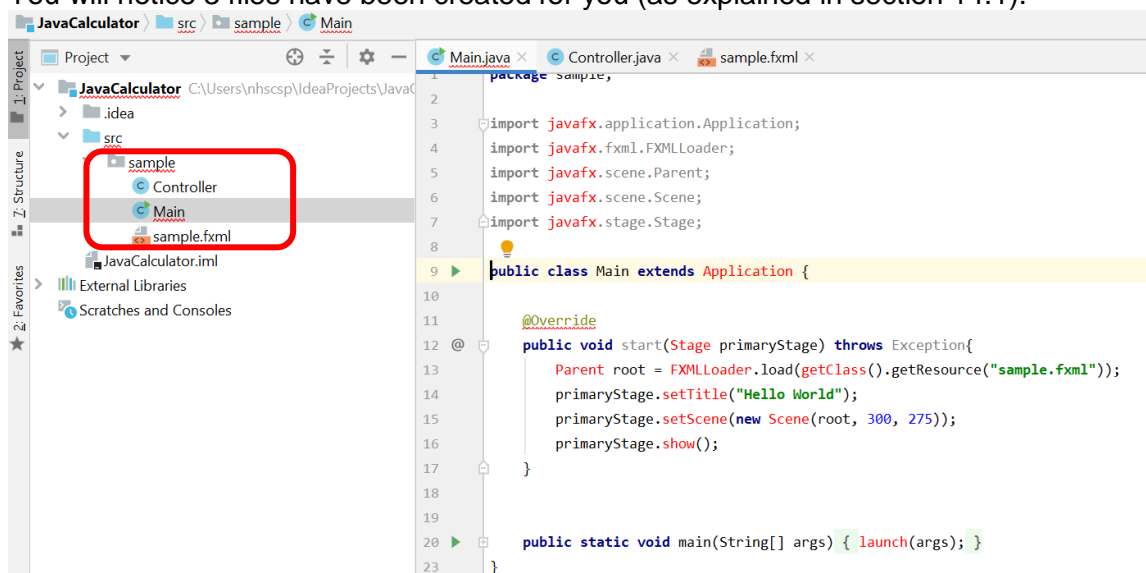


14.3.1 Creating a new JavaFX Project in IntelliJ

1. First, select File > New > Project.
2. Select **JavaFX** on the left panel and click Next.



3. Next, give your project a name (e.g. JavaCalculator) and click Finish. You will notice 3 files have been created for you (as explained in section 14.1).



4. Import the JavaFX SDK as per steps 6 and 7 on page 9 and 10 of Chapter 13 notes. This will resolve all the errors seen in step 3 above.

5. Add in a new module-info.java. The code as follows:

```
module JavaCalculator {
    requires javafx.controls;
    requires javafx.fxml;
    opens sample;
}
```

Note that for this project, we also requires javafx.fxml.

6. Update the title of the stage, and remove the width and height declaration.

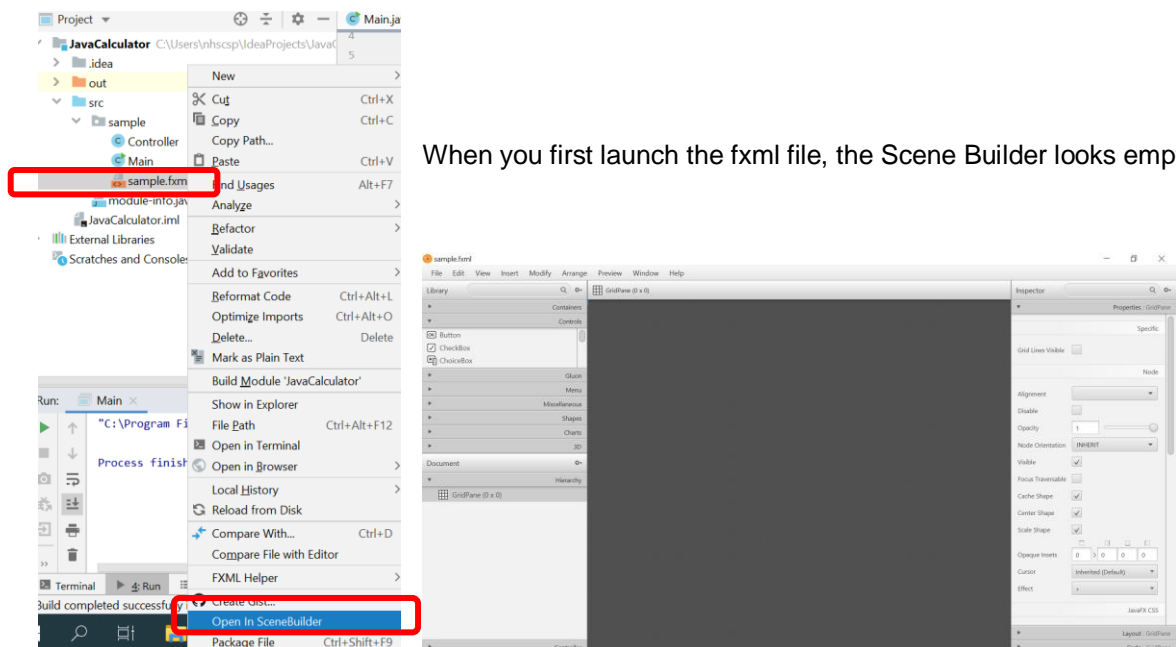
```
primaryStage.setTitle("Java Calculator");
primaryStage.setScene(new Scene(root));
```

Run your Main class. You will see a default blank frame with a updated title.

14.3.2 Designing the UI via Scene Builder

The FXML document in your project is very much like HTML and you can write code to “describe” the layout of the application. Now let’s design the user interface (UI) of our project using Scene Builder.

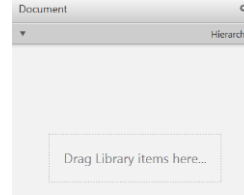
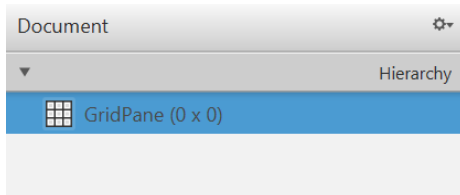
7. With Scene Builder properly installed and path set correct, right click on the fxml file > Select Open in Scene Builder.



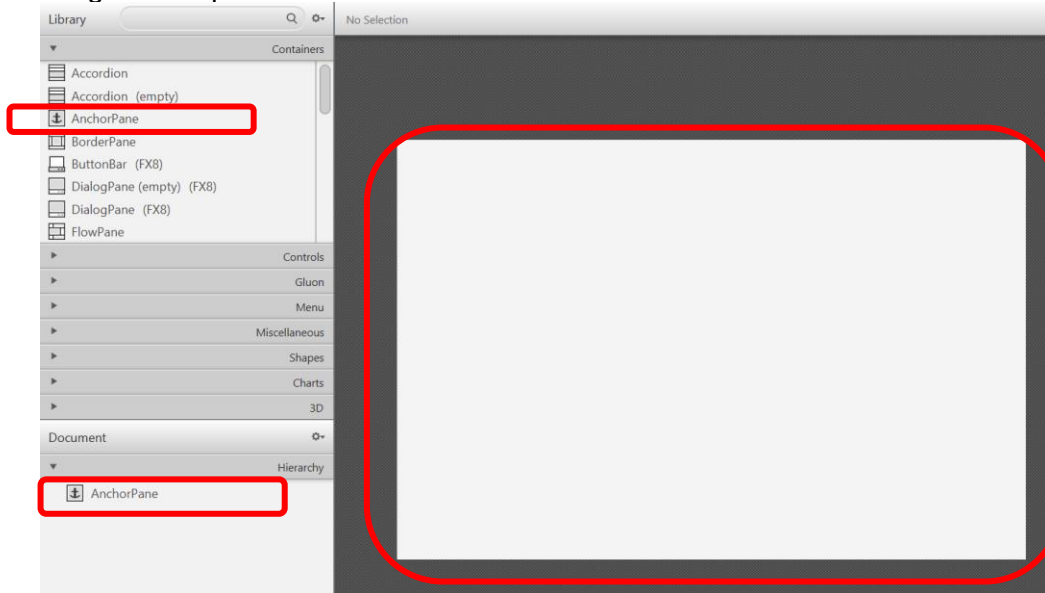
If you are unable to open the fxml file in Scene Builder, refer to Lab 13.1 Q2 and Q3 to do the settings.

8. You can then design your GUI by dragging and dropping controls (e.g. buttons) into the frame (i.e. scene).

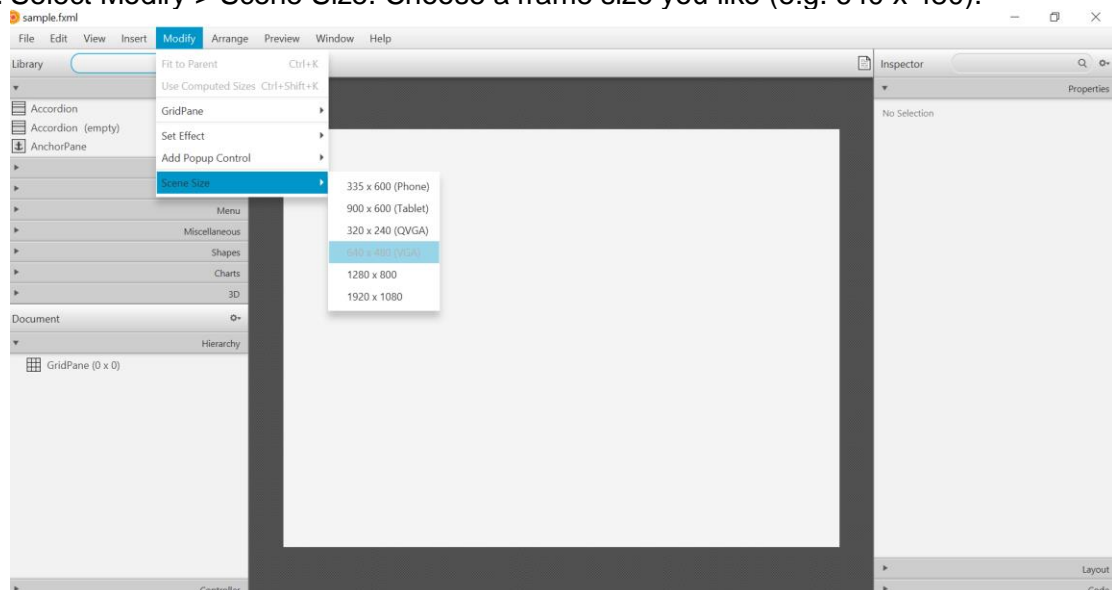
First, delete the default GridPane in the Hierarchy Panel. You may do so by selecting GridPane in the Document Panel. Press the “Delete” key on your keyboard to delete. Your Hierarchy panel will be empty after the delete (figure on the right).



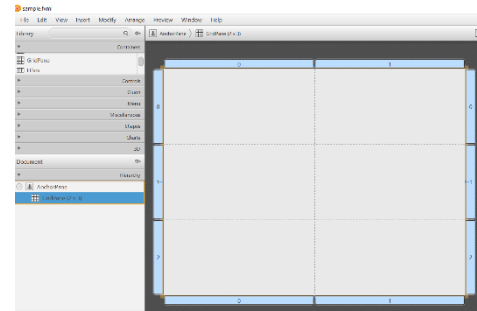
9. Drag and drop in an AnchorPane from the Container Panel.



10. Select Modify > Scene Size. Choose a frame size you like (e.g. 640 x 480).

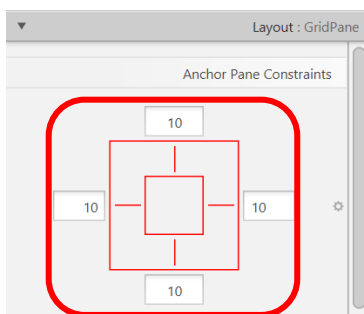
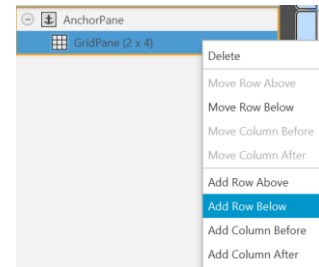


11. Add a GridPane inside the AnchorPane.
Next, select Modify > Fit to Parent



12. Add in another row in the GridPane by right-clicking on the GridPane > GridPane > Add Row Below.

In the Layout Panel, set the cellpadding of each side to 10.



13. In the Controls panel you will find Buttons, Labels and TextFields. Drag out the controls we need into the different cells.

For the group of buttons (+ - * / ^), you can use a ButtonBar Container to layout.

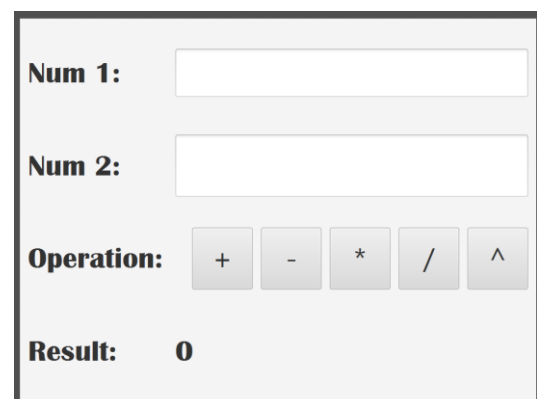
Set font size of all to 36px Britannic Bold.

You may make this change via the Properties Panel.

Quick Tip: You can select all the labels, textfield and buttons you wish to have the font changed and change the font at the same time. Hold down the Shift key while you select each control.

You may modify the text of each control by double clicking it.

Once done, you should get a UI as shown on the right.



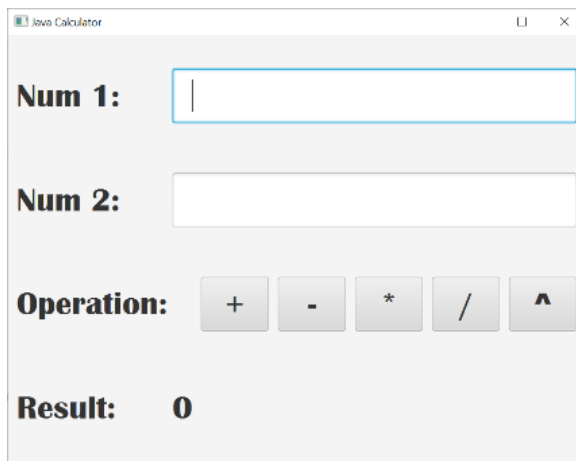
14. Under Code > fx:id, name the following controls as such:

Control	fx:id
Num 1 Textfield	num1TF
Num 2 Textfield	num2TF
Results label	resultLbl
+ button	plusBtn
- button	minusBtn
* button	timesBtn
/ button	divideBtn
^ button	powerBtn

This step will give each of the control above a variable name which you may refer to in your java code later in Controller.java.

15. Save the changes (File > Save) and close Scene Builder. Compile and run your project. You will see the UI of the calculator.

Note that nothing will happen when you click on the button now as you have not added any event-handling code!

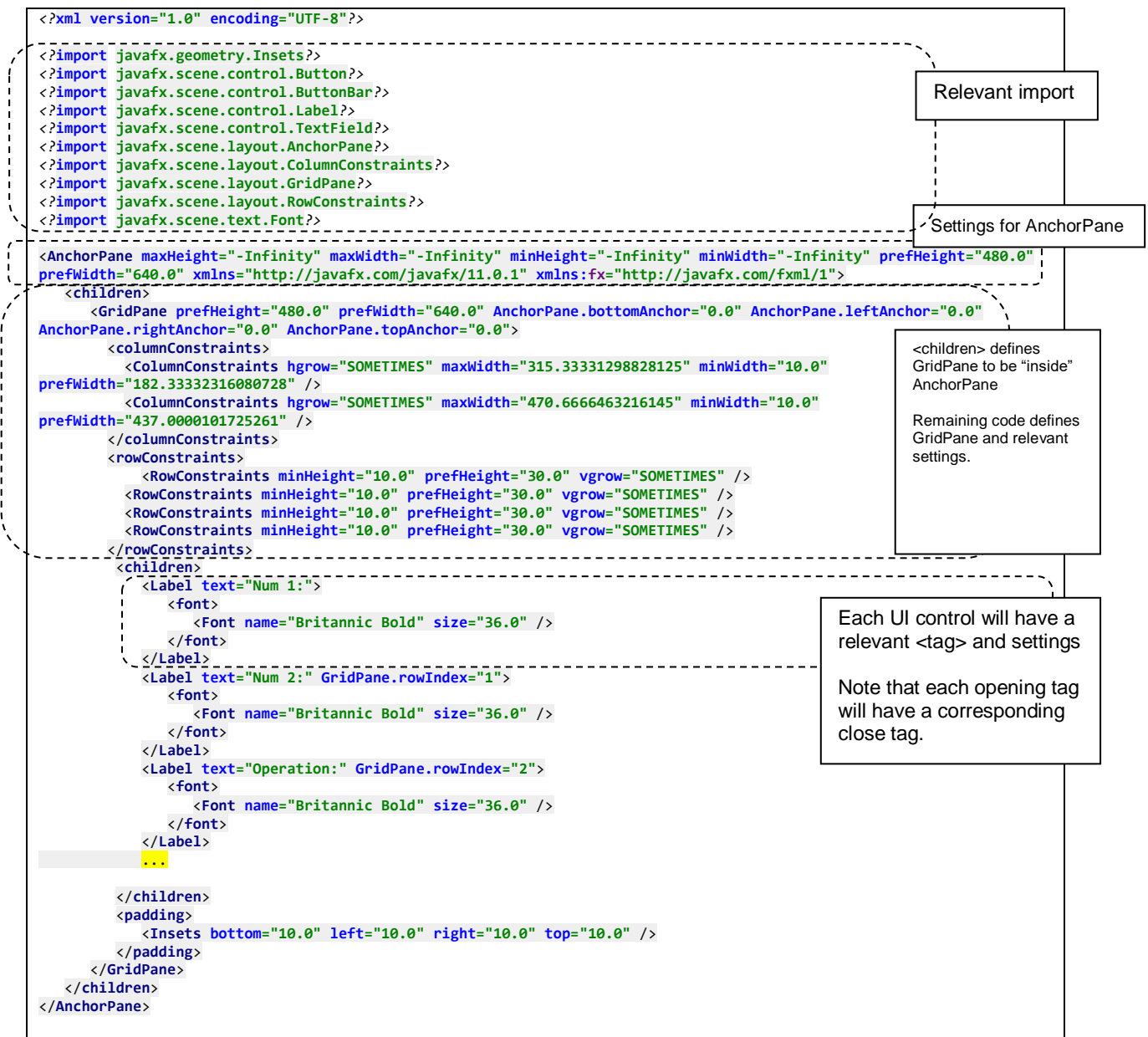


14.3.3 Understanding the FXML code

FXML is an XML-based language that provides the structure for building a user interface separate from the application logic of your code. While the FXML code can be generated using Scene Builder, it's also possible to code it yourself from scratch.

Having some understanding of the FXML code will allow you to do simple edits more easily.

Part of the FXML code for simple calculator application is shown below:



!!! Word of Caution !!!

Generally, it is recommended to leave the generation of the FXML code to Scene Builder.

However, if you are familiar and comfortable with FXML code, it may be convenient to edit the FXML code directly for simple tasks like adding `fx:id` and registering event.

Do be careful when you edit the FXML code directly as any small typo can break the code.

14.3.4 Event Handling Using Controller Class

In *event-driven programming*, code is executed upon **activation of events**. An *event* can be defined as a type of signal to the program that something has happened. The event is generated by external user actions such as mouse movements, mouse button clicks, and keystrokes, or by the operating system, such as a timer.

If a component is meant to be able to respond to some user action (i.e. event) then the component needs to specify what objects are listeners that will respond to events fired by that component. This is known as **registering the listener**. Finally, the programmer must define the methods (i.e. the eventhandler) that will be invoked when the event is sent to the listener.

For example, in the HelloWorldFXCode in Chapter 13, the code:

```
//connect btn to the code that runs when you click it
btn.setOnAction(e -> buttonClick(e));
```

registers the eventhandler buttonClick() to a Button named btn.

The eventhandler code is then given in the buttonClick() method:

```
//code to be triggered when user clicks on the button
public void buttonClick(ActionEvent event) {
    System.out.println("You clicked me!");
    label.setText("Hello World!"); //change text of Label. Note that you are able
                                //to access Label in this method as it's instance var
}
```

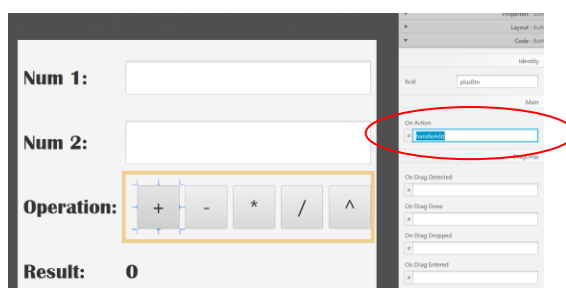
The FXML allows you to design the graphical user interface (GUI) using Scene Builder. The Main.java helps to launch your application program. **The eventhandling is done via a Controller class.** The Controller class allows something to happen when we interact with the UI.

IntelliJ allows us to generate a controller class via the FXML file created (if you have installed the plugin “FXML Helper” correctly). This will save us some time in typing some code (which can be generated by IntelliJ).

Event handling in FXML and Scene Builder

In a JavaFX project, the event handling code is added in the Controller class, which can be partly generated by IntelliJ. **However, before we generate the source code for the controller file, we should determine which controls need an event-handler in Scene Builder, and name the controls and the relevant event-handler first.**

- For each button in the calculator app, add in the name of the eventhandler under the Code Panel (the name is just a method name, you can name it anything).

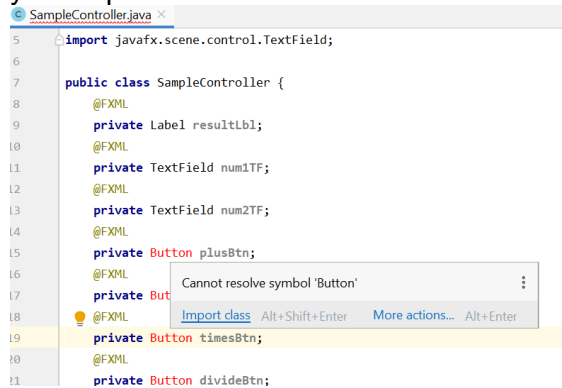


Control	On Action
+ button	handlePlus
- button	handleMinus
* button	handleTimes
/ button	handleDivide
^ button	handlePower

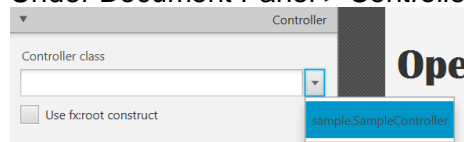
Save your work.

17. Now right click on the FXML document, FXML Helper > Generate a Controller > Private access level A new file named SampleController.java has been created for you.

Double click on the SampleController.java file. You will see some code generated for you. Import relevant class to resolve the errors.



18. Link the FXML file to the controller via the Controller panel in Scene Builder. Under Document Panel > Controller:



19. You may now add in the code required for each button event as follows:

```
@FXML
public void handlePlus(ActionEvent event) {
    double num1 = Double.parseDouble(num1TF.getText());
    double num2 = Double.parseDouble(num2TF.getText());
    double ans = num1 + num2;
    resultLbl.setText(ans+"");
}

@FXML
public void handleMinus(ActionEvent event) {
    double num1 = Double.parseDouble(num1TF.getText());
    double num2 = Double.parseDouble(num2TF.getText());
    double ans = num1 - num2;
    resultLbl.setText(ans+"");
}

@FXML
public void handleTimes(ActionEvent event) {
    double num1 = Double.parseDouble(num1TF.getText());
    double num2 = Double.parseDouble(num2TF.getText());
    double ans = num1 * num2;
    resultLbl.setText(ans+"");
}

@FXML
public void handleDivide(ActionEvent event) {
    double num1 = Double.parseDouble(num1TF.getText());
    double num2 = Double.parseDouble(num2TF.getText());
```

```

    double ans = num1 / num2;
    resultLbl.setText(ans+"");
}

@FXML
public void handlePower(ActionEvent event) {
    double num1 = Double.parseDouble(num1TF.getText());
    double num2 = Double.parseDouble(num2TF.getText());
    double ans = Math.pow(num1, num2);
    resultLbl.setText(ans+"");
}

```

You should be familiar with the event-handling code above. Something new will be the @FXML annotation.

The @FXML annotation enables the FXMLLoader (in Main.java) to inject values defined in an FXML file (in sample.fxml) into references in the controller class (in SampleController.java).

For example, in Main.java, the line belows load the sample.fxml file.

```
Parent root = FXMLLoader.Load(getClass().getResource("sample.fxml"));
```

In SampleController.java, add in the @FXML annotation.

```

@FXML
private Label resultLbl;

```

In sample.fxml, the fx:id will point to resultLbl:

```
<Label fx:id="resultLbl" text="0" GridPane.columnIndex="1" GridPane.rowIndex="3">
```

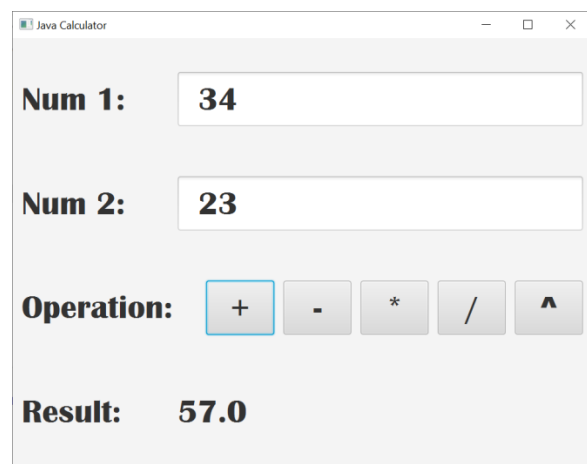
When the FXMLLoader loads the FXML file, it will attempt to inject any elements that have an fx:id attribute into the controller. It will look for:

- Any public field with a variable name matching the fx:id attribute, or
- Any field (public or not) with a variable name matching the fx:id attribute that is annotated with @FXML.

It is good practice to make your fields private (encapsulation), then each declaration must be annotated @FXML for the injection to work.

Now you can compile and run your program.

Congrats! Your Java Calculator is completed!



Handling multiple events with ONE event-handler

In the previous section, we coded an event-handler for each button. However, it is also possible to just use ONE event-handler to handle all the button click events.

In this approach, all 5 buttons will share the SAME event handler method. The code required is shown below:

```
@FXML
private void handleButtonAction(ActionEvent event) {
    double num1 = Double.parseDouble(num1TF.getText());
    double num2 = Double.parseDouble(num2TF.getText());
    double ans;
    if(event.getSource() == plusBtn){
        ans = num1 + num2;
    }
    else if(event.getSource() == minusBtn){
        ans = num1 - num2;
    }
    else if(event.getSource() == timesBtn){
        ans = num1 * num2;
    }
    else if(event.getSource() == divideBtn){
        ans = num1 / num2;
    }
    else{
        ans = Math.pow(num1, num2);
    }
    resultLbl.setText(ans+"");
}
```

event.getSource() returns the object that triggers the event.

Hence by comparing which button object event.getSource() returns, we will know which of the 5 buttons was being clicked by user.

Note that you need to register event-handler `handleButtonAction()` for all 5 buttons via Scene Builder.

Or you can edit the FXML file directly (see below):

```
<ButtonBar prefHeight="89.0" prefWidth="312.0" GridPane.columnIndex="1" GridPane.rowIndex="2">
  <buttons>
    <Button fx:id="plusBtn" contentDisplay="CENTER" mnemonicParsing="false" onAction="#handleButtonAction" text="+">
      <font>
        <Font size="36.0" />
      </font>
    </Button>
    <Button fx:id="minusBtn" contentDisplay="CENTER" mnemonicParsing="false" onAction="#handleButtonAction" text="-">
      <font>
        <Font size="36.0" />
      </font>
    </Button>
    <Button fx:id="timesBtn" contentDisplay="CENTER" mnemonicParsing="false" onAction="#handleButtonAction" text="*">
```

Which approach is “better” for this particular example? Why?

For this particular example, the code will be shorter if we use one event-handler as the code to calculate are largely the same. Using this approach reduces code duplication and makes maintenance of code easier.

How do you decide which approach to use?

You should generally use one event-handler if the buttons function largely the same. Multiple event-handler can be used if the functionality of the buttons are very different and there will not be much code duplication between the event handlers.

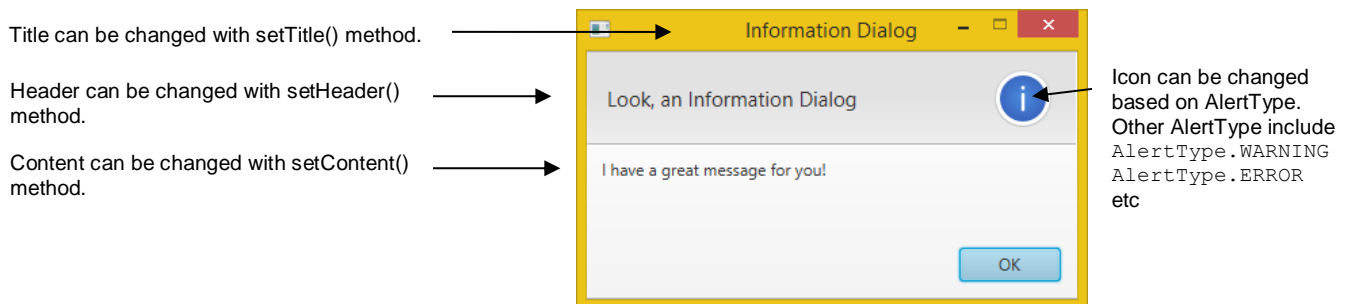
Of course, if the controls are different, they should NOT share the same event-handler (e.g. a checkbox event cannot be grouped with button click events).

14.4 Other Useful Features

14.4.1 Adding Dialog Boxes

The graphical control element dialog box is a small window that communicates information to the user and prompts them for a response.

For example, an Information Dialog in JavaFX is shown below:



The code required to show a dialog box is given below:

```
Alert alert = new Alert(AlertType.INFORMATION);
alert.setTitle("Information Dialog");
alert.setHeaderText("Look, an Information Dialog");
alert.setContentText("I have a great message for you!");
alert.showAndWait();
```

You can read about other types of Dialog box at the following website:

<http://code.makery.ch/blog/javafx-dialogs-official/>

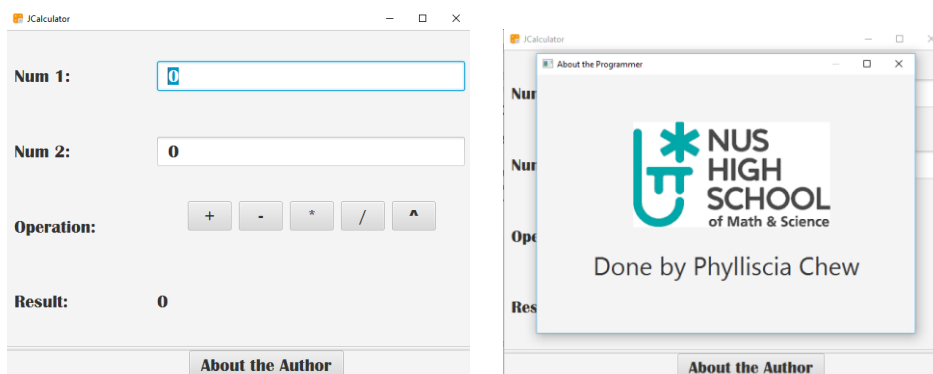
14.4.2 Adding a Pop-up Window Frame

The dialog boxes are useful if you want to popup simple messages to user. However sometimes you may want to popup a window that you design.

For example, if we would like to let people know more about the programmer of the app, we can do this using an “About the Programmer” window (i.e. code a new stage for this).

In the jCalculator app, we can add a new Button “About the Author”. Upon clicking the button, a new window (stage) will pop up showing information about the programmer.

Note that you need to add and design a new FXML file for the new window.



The code required to popup a new stage is shown below. Add this code in the controller java file, under the new button event-handler.

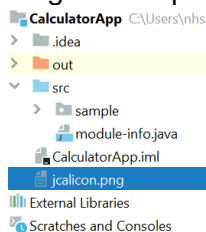
```
@FXML
private void handleAbout(ActionEvent event) {
    try {
        Stage stage = new Stage();
        Parent root = FXMLLoader.Load(getClass().getResource("About.fxml"));
        stage.setScene(new Scene(root));
        stage.setTitle("About the Programmer");
        stage.initModality(Modality.WINDOWS_MODAL);
        stage.initOwner(aboutBtn.getScene().getWindow());
        stage.showAndWait();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Refer to Chapter 13, page 12, for different types of modality.

14.4.3 Loading a Program Icon

You can set the icon for your JavaFX program by following the steps below:

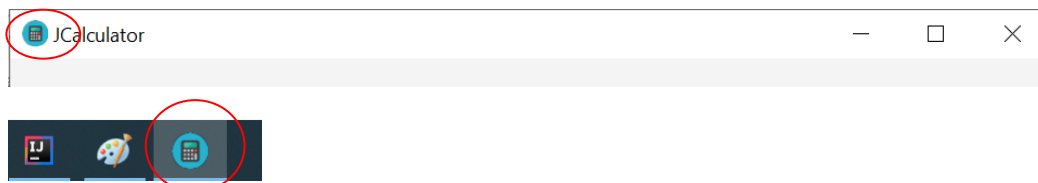
1. Drag and drop the icon image file to your JavaFX project:



2. Add the code below to load the icon resource. The code should be added to the start() method in Main.java:

```
// Set the application icon.
primaryStage.getIcons().add(new Image("file:jcalicon.png"));
```

Note that the icon image will be shown on the title bar of your JavaFX program, and also as an icon on your taskbar.



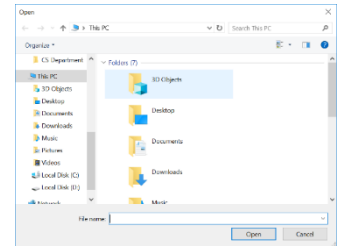
14.4.4 Using a JavaFX File Dialog

Recall that in Chapter 11, you learnt how to create a file dialog with JFileChooser. Here, we will go through how to create file dialog with JavaFX.

To create an Open File Dialog like the one on the right:

Use the following code in an eventhandler:

```
@FXML
public void handleButtonClick(ActionEvent event) {
    // TODO Autogenerated
    FileChooser fileChooser = new FileChooser();
    File selectedFile = fileChooser.showOpenDialog(null);
    if (selectedFile != null) {
        label1.setText("File selected: " + selectedFile.getName());
    }
    else {
        label1.setText("File Selection Cancelled!");
    }
}
```



Add in relevant file IO code if you wish to read the contents of the file.

Similarly, if we wish to create a Save Dialog, use a “Save Dialog”:

```
File savedFile = fileChooser.showSaveDialog(null);
```

14.5 Other Common Controls

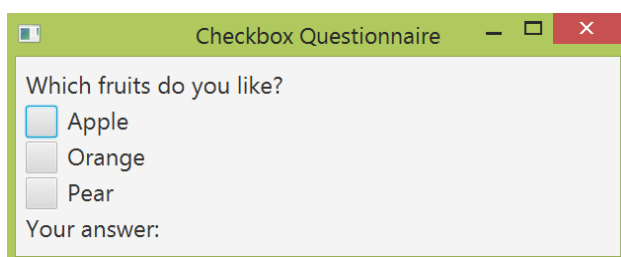
We have explored the 3 main controls so far (Button, TextField and Labels). Now that you have a good understanding of how to create GUI programs using JavaFX FXML, using the other controls follow quite a similar approach.

This section we will explore briefly the use of other common controls.

14.5.1 CheckBox

Checkboxes are handy little widgets that allow you to select one or more options at the same time. You can either have your program watch for changes in a user selection right away (i.e. when the user clicks on or off a single checkbox) or in a separate button (like a ‘done’ button for example).

You can create a simple questionnaire like the one below using labels and checkbox control in Scene Builder.



Once you are done with the GUI design with Scene Builder, generate the controller file.

Write the code for the event handler in the controller java file.

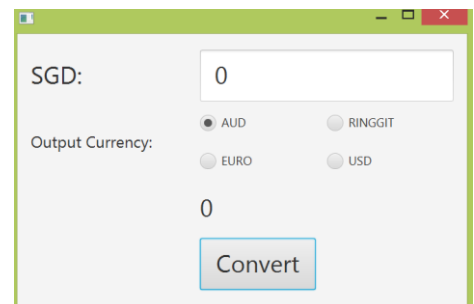
For example:

```
@FXML private void handleCheckAction(ActionEvent event) {
    String ans = "Your answer: ";
    int count = 0;
    if(appleCB.isSelected()){ //checks if the checkbox is selected
        ans += appleCB.getText() + ", ";
        count++;
    }
    if(orangeCB.isSelected()){
        ans += orangeCB.getText() + ", ";
        count++;
    }
    if(pearCB.isSelected()){
        ans += pearCB.getText() + ", ";
        count++;
    }
    ans = ans.substring(0,ans.length()-2) + ". You selected " + count + " fruits.";
    ansLbl.setText(ans);
}
```

14.5.2 RadioButton

Radio buttons are like the brother to CheckBoxes, the only difference is that they can be setup to force the user to select ONLY ONE of them at a time. All settings and coding for RadioButton is the same as that with CheckBoxes. However, as RadioButton forces user to select only one at a time, you need to follow some additional steps.

1. Create the following GUI with Scene Builder.



2. In the controller java file, add in code to create a ToggleGroup object.

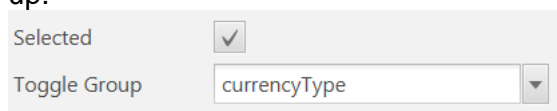
For example:

```
@FXML private ToggleGroup currencyType;
```

3. In Scene Builder, select the RadioButton.

Under the Properties Panel of the RadioButton, set the ToggleGroup to the one you declared in step 2.

You can also choose if you would like to set this RadioButton to be "Selected" on start up.



Repeat the same for each RadioButton.

The purpose of the ToggleGroup is to **logically group the RadioButtons together**.

Only one RadioButton per ToggleGroup can be selected.

14.5.3 ComboBox

A combobox is essentially a dropdown list. For example:
The code for ComboBox:

```
import java.net.URL;
import java.util.ResourceBundle;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.*;

public class FXMLDocumentController implements Initializable {
    ObservableList<String> fruits = FXCollections.observableArrayList("Apple", "Pear", "Banana", "Grapes");
    @FXML private ComboBox<String> combo;
    @FXML private Label label;

    @FXML
    private void handleSelectAction(ActionEvent event) {
        label.setText("You have selected " + combo.getValue());
    }

    @Override
    public void initialize(URL url, ResourceBundle rb) {
        combo.setItems(fruits);
        //combo.setValue("Pear");
    }
}
```

