COMP3015 Data Communication and Networking
GUI

JavaFX
JavaFX is one of the libraries for building Java graphical user interfaces. It was part of Java but was removed after Java 10. If you want to use JavaFX in Java 11 or later, you can visit its official website https://openjfx.io/.
In this lab, we use Java 8, so we do not need to install JavaFX additionally.

Building Java GUI Application
To build a Java GUI application, we need a public class that inherits the Application class (`javafx.application.Applicatoin`) and implement its abstract method – `start()`. The following is a program code to build an empty window:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class MyApp extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        BorderPane root = new BorderPane();
        primaryStage.setScene(new Scene(root, 300, 300));
        primaryStage.setTitle("My 1st App");
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

In the `start()` method, we do:

1. Create a root pane.
2. Create a scene with the root pane and add the scene to the primary stage. You can think the primary stage is the window.
3. Set the title to the window.
4. Show the window.

Containers
A container is used to store other components. JavaFX has different containers that arrange their child components. The following are the common used containers:

BorderPane
It puts its children to TOP, LEFT, CENTER, RIGHT, and BOTTOM.

AnchorPane

It puts its children depending on their position settings.

FlowPane

It puts its children from the left to the right, and from the top to the bottom.

VBox

It arranges its children from the top to the bottom.

HBox

It arranges its children from the left to the right.

GridPane

It puts its children in rows and columns.

ScrollPane

It provides vertical and horizontal scroll bars. But it can have one child only.

Controls

Controls are used to display information or capture user inputs. The common used controls include:

Label

It is used to display text.

ImageView

It is used to display an image.

TextField

It allows the user to input text.

PasswordField

Similar to the `TextField` control but you cannot see what you type.

Button

It triggers an event if the user clicks it.

CheckBox

It allows to be checked or unchecked.

ComboBox

It provides a pop-up list for the selection.

RadioButton

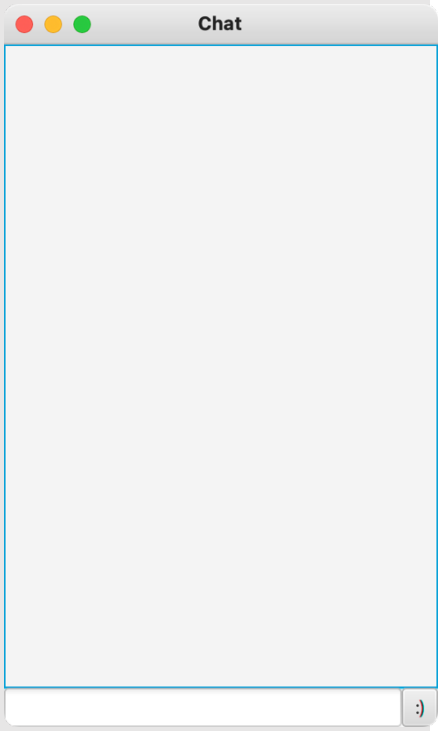
A group of the `RadioButton` controls is used to form a single-selection multi-choice design.

Simple Chat UI

Combined with the containers and controls, we can create some complicated UI. But, before you do so, let's learn from the simple. Next, we are going to create an UI for a chat application.

```
...
public class ChatUI1 extends Application {

    ObservableList<Node> children;
    int msgIndex = 0;

    @FXML private ScrollPane scrollPane;
    @FXML private TextField txtInput;

    ChatUI2 chat;

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Chat");

        VBox messagePane = new VBox();
        children = messagePane.getChildren();

        scrollPane = new ScrollPane();
        scrollPane.setContent(messagePane);
        scrollPane.setFitToWidth(true);
        scrollPane.setFitToHeight(true);

        ...

        BorderPane root = new BorderPane();
        root.setCenter(scrollPane);
        primaryStage.setScene(new Scene(root));
        primaryStage.setMinWidth(300);
        primaryStage.setMinHeight(500);

        txtInput = new TextField();
        ...

        Button btnEmoji = new Button(":)");
        ...

        BorderPane container = new BorderPane();
        container.setCenter(txtInput);
        container.setRight(btnEmoji);

        root.setBottom(container);

        primaryStage.show();
   }
   ...

    public static void main(String[] args) {
        launch(args);
    }
}
```

Check `ChatUI1.java` for the details



The program segment above does the followings:

- Use a border pane to be a root pane.
- Put a scroll pane into the center region of the root pane.
- Put a VBox into the scroll pane. The VBox acts as a message pane to contain and display messages.
- Create a second border pane that is used to contain a text field and a button. The text field is in its center region, and the button is in its right region.
- Then put the second border pane to the bottom region of the root pane.

You may discover that variable `txtInput` and variable `scrollpane` are declared outside the `start()` method but variables for other containers and controls are declared locally in the `start()` method. It is because we will define other methods later to access the text field and the scroll pane.

Interaction and Event Handling

To make your program interact with the user, we need to add the event handler to some controls. For example, your program display a message when the user types something in the text field and then presses ENTER.

```
        txtInput.setOnKeyPressed(event->{
            if (event.getCode().toString().equals("ENTER"))
                displayMessage();
        });
```

Or, your program display an emoji image when the user clicks the button.

```
        btnEmoji.setOnMouseClicked(event->{
            displayEmoji();
        });
```

Or, even the scroll pane scrolls to the bottom automatically when the height of the message pane is changed.

```
        messagePane.heightProperty().addListener(event->{
            scrollPane.setVvalue(1);
        });
```

Displaying Message

To display the message, we can simply use the label control. By default, the width of the label depends on its content and is left-aligned when added to the message pane (VBox). We need messages to go to the left and right to represent sent messages and received messages, respectively.

Therefore, we wrap a label inside an HBox and assign an alignment to the HBox.

```
    private Node messageNode(String text, boolean alignToRight) {
        HBox box = new HBox();
        box.paddingProperty().setValue(new Insets(10, 10, 10, 10));

        if (alignToRight)
            box.setAlignment(Pos.BASELINE_RIGHT);
        Label label = new Label(text);
        label.setWrapText(true);
        box.getChildren().add(label);
        return box;
```

```
    }
```

When a message Node is ready, we add it to the message pane. The `Platform.runLater()` block is used to ensure that the JavaFX thread executes the UI code correctly. If your program involves multithreading and you want to update the UI through the child thread, you must use the `Platform.runLater()` block.

```
    private void displayMessage() {
        Platform.runLater(() -> {
            String text = txtInput.getText();
            txtInput.clear();
            children.add(messageNode(text, msgIndex == 0));
            msgIndex = (msgIndex + 1) % 2;
            scrollPane.setVvalue(1);
        });
    }
```

Displaying Image

We use similar logic to display images, but instead of labels, we use image views. In order to display the image, we need a file input stream to open the image file and read its contents. Then, we pass the content to the image object and wrap the image object in an image view. We can resize the image by setting the width and height of the image view. Finally, we wrap the image view in a message pane.

```
    private Node imageNode(String imagePath, boolean alignToRight) {
        try {
            HBox box = new HBox();
            box.paddingProperty().setValue(new Insets(10, 10, 10, 10));

            if (alignToRight)
                box.setAlignment(Pos.BASELINE_RIGHT);
            FileInputStream in = new FileInputStream(imagePath);
            ImageView imageView = new ImageView(new Image(in));
            imageView.setFitWidth(50);
            imageView.setPreserveRatio(true);
            box.getChildren().add(imageView);
            return box;
        } catch (IOException ex) {
            ex.printStackTrace();
            return messageNode("!!! Fail to display an image !!!", alignToRight);
        }
    }
```

```
    private void displayEmoji() {
        Platform.runLater(() -> {
            children.add(imageNode("emoji.png", msgIndex == 0));
            msgIndex = (msgIndex + 1) % 2;
            scrollPane.setVvalue(1);
        });
    }
```

FXML – Loading UI Setting from FXML file

In the previous example, we found that the code for building the UI is quite long even if the UI is simple. JavaFX allows separating the UI code from the Java logic. FXML is a mark-up language for describing the UI. The content looks like a XML code with tags and attributes.

```
<BorderPane  xmlns:fx="http://javafx.com/fxml" maxHeight="-Infinity"
            maxWidth="-Infinity" minHeight="-Infinity" minWidth="-Infinity"
            prefHeight="400.0" prefWidth="300.0" >
   <center>
      <ScrollPane fx:id="scrollPane" fitToHeight="true" fitToWidth="true"
            prefHeight="374.0" prefWidth="460.0" BorderPane.alignment="CENTER">
         <content>
            <VBox fx:id="messagePane" />
         </content>
      </ScrollPane>
   </center>
   <bottom>
      <BorderPane prefHeight="0.0" prefWidth="0.0" BorderPane.alignment="CENTER">
         <center>
            <TextField fx:id="txtInput" BorderPane.alignment="CENTER" />
         </center>
         <right>
            <Button fx:id="btnEmoji" mnemonicParsing="false" text=":)"
                        BorderPane.alignment="CENTER" />
         </right>
      </BorderPane>
   </bottom>
</BorderPane>
```

With FXML, we can modify the `start()` method as follows:

```
    @Override
    public void start(Stage primaryStage) throws IOException {
        FXMLLoader loader = new FXMLLoader(getClass().getResource("ChatUI2.fxml"));
        loader.setController(this);
        Parent root = loader.load();
        Scene scene = new Scene(root);
        primaryStage.setScene(scene);
        primaryStage.setTitle("Chat");
        primaryStage.setMinWidth(300);
        primaryStage.setMinHeight(500);
        primaryStage.show();
    }
```

We use the FXML loader to load the UI setting from the ChatUI2.fxml file, so the `start()` method becomes shorter and clearer. But, we need to move all event handling code to another method – `initialize()`.

```
    @FXML
    private ScrollPane scrollPane;
    @FXML
    private TextField txtInput;
    @FXML
    private Button btnEmoji;
    @FXML
    private VBox messagePane;

    @FXML
    protected void initialize() {
        children = messagePane.getChildren();
        messagePane.heightProperty().addListener(event -> {
            scrollPane.setVvalue(1);
        });

        txtInput.setOnKeyPressed(event -> {
            if (event.getCode().toString().equals("ENTER"))
                displayMessage();
```

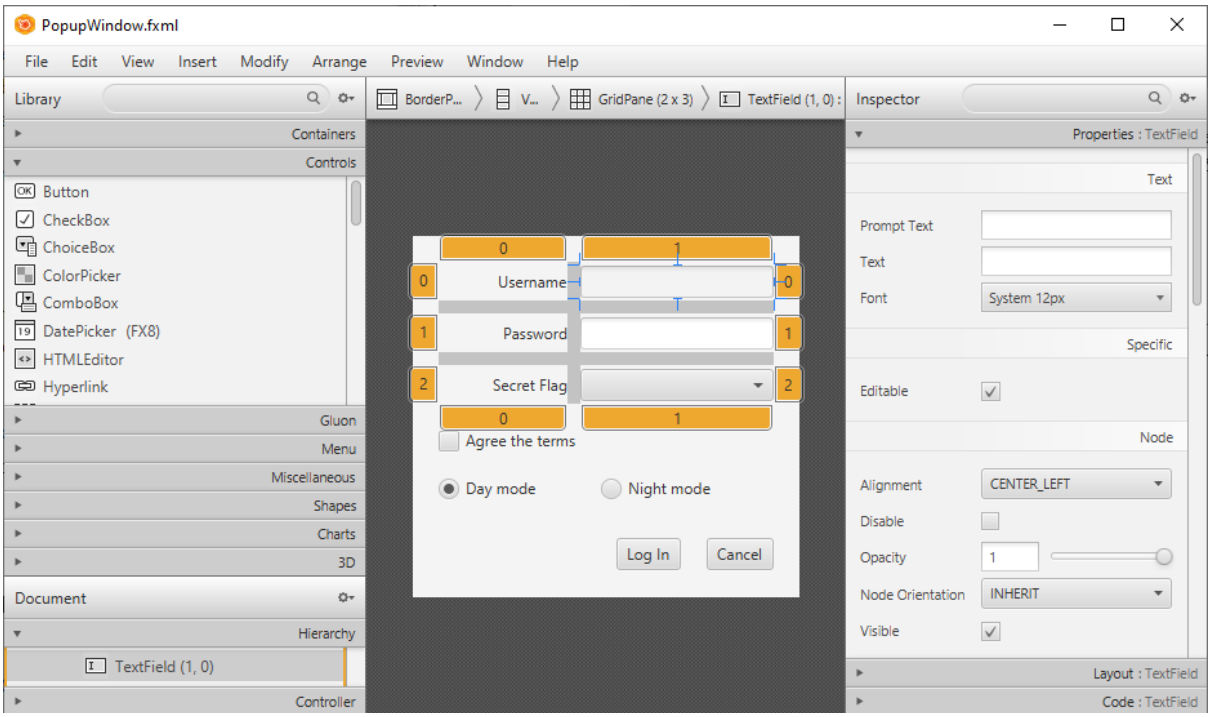
Check `ChatUI2.java` for the details

```
        });

        btnEmoji.setOnMouseClicked(event -> {
            displayEmoji();
        });
    }
```

After looking at the code for `ChatUI2.java`, you should see that there are no statements that assign values to object variables declared with `@FXML` mark. This is because once you put the `@FXML` mark at the beginning of a variable, the FXML loader will automatically map it. Of course, the variable names must match the components described in the FXML file.

Scene Builder

However, building UI with FXML is still time-consuming. Of course, we better have a tool to build the UI by using "drag and drop". Scene Builder is a free tool that can help. You can download it from the following URL - https://gluonhq.com/products/scene-builder/.



After saving the FXML file, you need to change it a bit before you can use it in your java program. You need to do the followings:

1. Put your FXML file together with your java file.

2. Delete the attribute about the controller from the root pane.

3. Delete all `xmlns:fx` attributes from the root pane.

4. Add the following attribute to the root pane:

```
xmlns:fx="http://javafx.com/fxml"
```

5. Save the changes of the fxml file.

Pop-Up Window

In the previous examples – ChatUI1 and ChatUI2, are single window applications. But the programs have multiple windows for many reasons, such as showing notifications, requesting user inputs, etc.
To create a pop-up window, we use the Stage class to create a new stage with the root pane, containers, and controls.

Consider the following code segment:

Check the following files:
- MainWindow.java
- PopupWindow.java

```
public class PopupWindow {
    Stage stage;

    public PopupWindow() throws IOException {
        stage = new Stage();
        FXMLLoader loader =
                new FXMLLoader(getClass().getResource("PopupWindow.fxml"));

        loader.setController(this);
        Parent root = loader.load();

        Scene scene = new Scene(root, 300, 280);
        stage.setScene(scene);

        stage.setTitle("Pop Up");

        stage.initModality(Modality.APPLICATION_MODAL);

        stage.setResizable(false);

        stage.showAndWait();
    }
}
```

The `initModality(Modality.APPLICATION_MODAL)` statement makes the pop-up window blocks the main window when it appears. So, the main window will not be active even if the user clicks it.

The `showAndWait()` statement blocks your program. Your program waits the user to close the pop-up window and then continue to run the next statement.

Useful Points about JavaFX

JavaFX is a large-scale library that provides many controls and containers. Each control/container has many properties for the customization, but it is impossible to cover all of them here. The followings are useful points about the JavaFX:

1. We can add an `OnMouseClicked` handler to any controls, such as `Label`. After adding the `OnMouseClicked` handler to a label, it works like a button. You can also try that on an `ImageView` control.

2. Each control/container has the `UserData` property that can be used to store an object. For example, you can store a file-path string or a file object in the `UseData` property of an ImageView control. So, you can get the information of the image rapidly.

   To do so, you only need to use the following statements to store and get the user data respectively:
   - `imageView.setUserData(data);`
   - `data = imageView.getUserData();`

3. If the root pane contains scroll panes, it is better to set the minimum height and minimum width of the primary stage to be larger than the height and width of the root pane.
   - `primaryStage.setMinHeight(height);`
   - `primaryStage.setMinWidth(width);`

4. If there is a scroll pane, it is better to make the vertical scrollbar always be shown by putting the following line in the `initialize()` method:

```
scrollPane.setVbarPolicy(ScrollBarPolicy.ALWAYS);
```