JavaFX

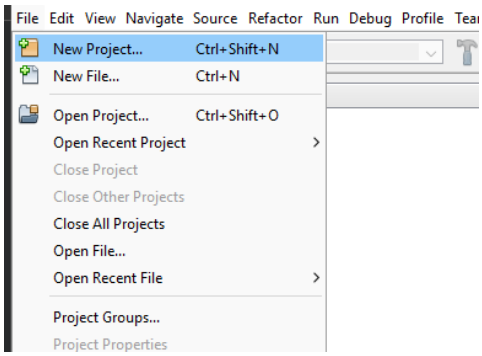
Overview:

JavaFX was designed with the MVC, or Model-View-Controller, pattern in mind. In a nutshell, this pattern keeps the code that handles an application's data separate from the UI code. Because of this, when we're using the MVC pattern, we wouldn't mix the code that deals with the UI and the code that manipulates the application data in the same class. The controller is sort of the middleman between the UI and the data.
When working with JavaFX, the model corresponds to the application's data model, the view is the FXML, and the controller is the code that determines what happens when a user interacts with the UI. Essentially the controller handles events, which we'll cover in a later lesson

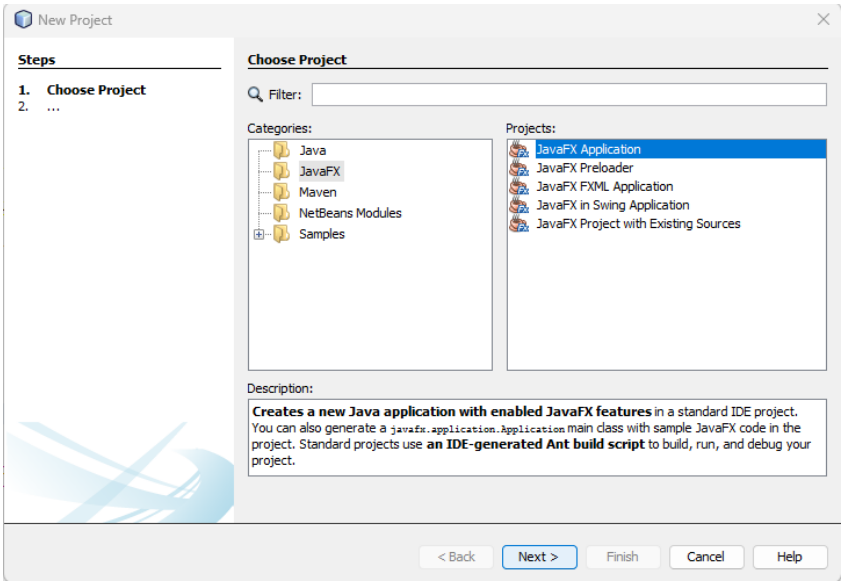
Setting Up the Environment:

To create a JavaFX Application in NetBeans 8.2 there are several steps to follow:

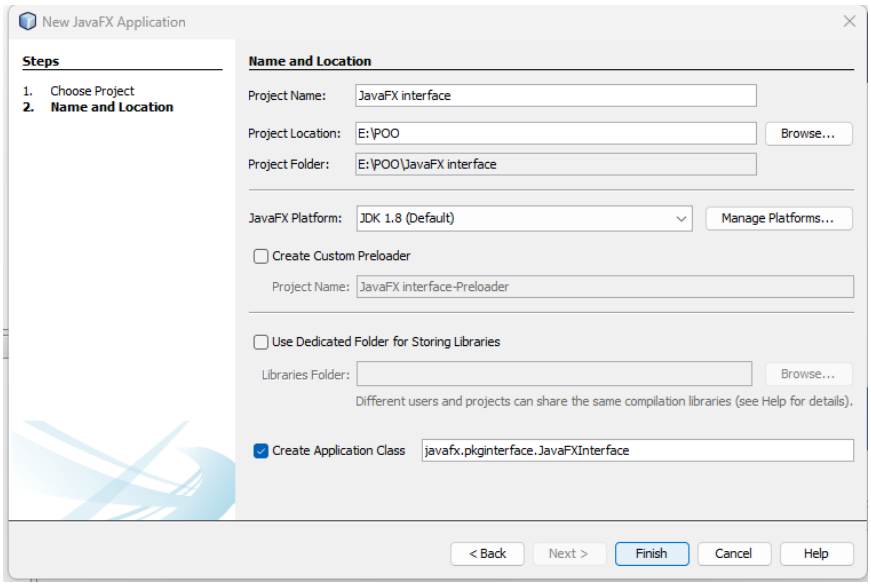
1. From the menu bar Click on “File” then select “New Project…”



2. Select "JavaFX" in the Categories and "JavaFX Application" in the Projects and then click “Next”.



3. Enter a project name and set the project location and then click “finish”

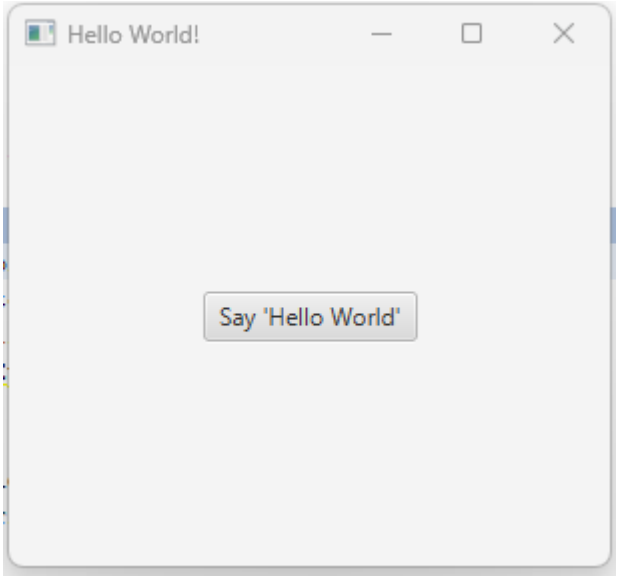


After completing these steps, a well-set-up project will appear in front of you, containing the following code:

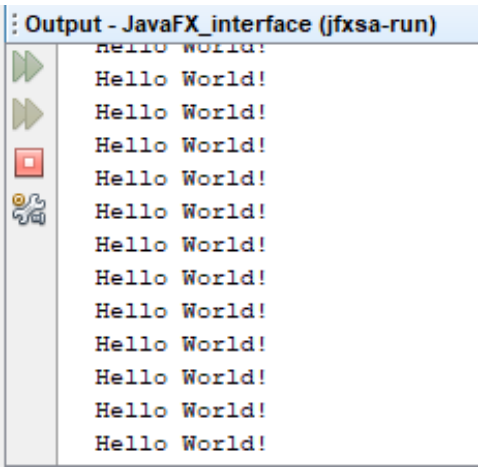
```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

/**
 *
 * @author Mokhtari Abdelheq
 */
public class Test extends Application {

    @Override
    public void start(Stage primaryStage) {
        Button btn = new Button();
        btn.setText("Say 'Hello World'");
        btn.setOnAction(new EventHandler<ActionEvent>() {

            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });

        StackPane root = new StackPane();
        root.getChildren().add(btn);

        Scene scene = new Scene(root, 300, 250);

        primaryStage.setTitle("Hello World!");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
```

```
launch(args);
```

Try building and running the code either by pressing F6 or by clicking on the 'Run' button located above; a window like that will appear



And by Clicking on the button a message of “Hello World” will show on your console



Congratulations! you made your first interaction with JavaFX.
PS: these steps may be variants from an IDE to another and from java version to another take your time to get familiar with the environment you are working in.

Your First Program in JavaFX:

In this section, we will guide you through the process of understanding and building your first JavaFX program from scratch, taking you step by step through each stage of development.

1. let's erase the existing code in the core of your program and start fresh with a clean slate for your main class

```
public class JavaFXInterface extends Application {
}
```

Here our main class extends the Application class which represents a fundamental class that serves as the entry point for JavaFX applications. It is part of the javafx.application package. The primary purpose of the Application class is to provide the structure and lifecycle management for a JavaFX application. However, it currently lacks the required **start** method.

2. The **start** method is where you define the main structure and behavior of your JavaFX interface next, we are going to define our **start** method inside our main class

```
@Override
 public void start(Stage primaryStage) throws Exception{
  }
```

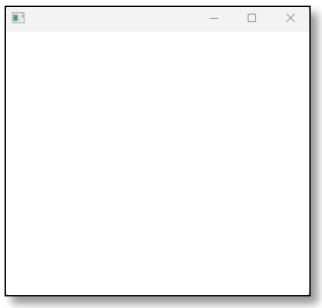
The **start** method is called when the JavaFX application is launched, and it takes a **Stage** object as a parameter. The **Stage** represents the main window or frame of the application.
`throws Exception` This part of the method signature indicates that the start method may throw an exception of type Exception We are going to talk more about it in future lessons

3. We need a **main** method besides our start method

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

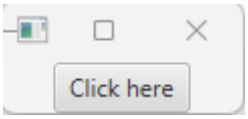
The **main** method, specifically the launch method within it, plays a crucial role in the execution of a JavaFX application. It kicks off the JavaFX platform, allows the **start** method in your application class to be invoked, and initializes the graphical user interface and other essential components needed for the JavaFX application to run. Without the **main** method with the launch call, your JavaFX application won't have a proper entry point and won't execute successfully.

4. Our next step is to begin adding our code inside the start method. The line `primaryStage.show();` is essential, as it displays an initially empty graphical interface.

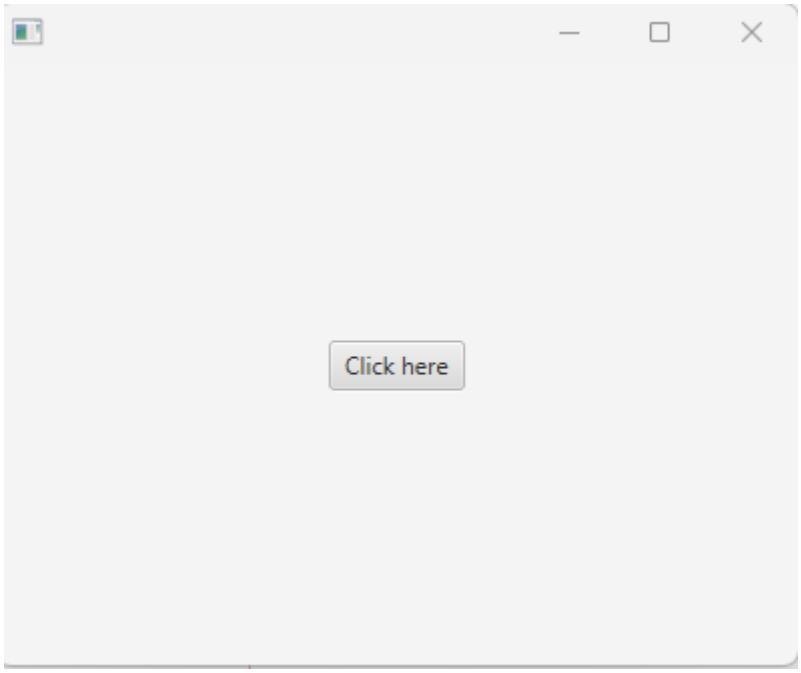


Then by initializing a button object from the Button Class `Button btn = new Button("Click here");` then Create a layout `StackPane root = new StackPane();` named root and then adding the button (btn) to this layout(root) using `root.getChildren().add(btn);` and then we create our scene from the Class Scene

`Scene scene = new Scene(root);` and adding the layout (root) to the scene object `primaryStage.setScene(scene);` by running the program a small window will appear



To create a bigger window, we are going to modify the object scene by invoking a different constructor `Scene scene = new Scene(root, 400, 300);`



The final code will be this way

```
public void start(Stage primaryStage) throws Exception{
            Button btn = new Button("Click here");

            StackPane root = new StackPane();
            root.getChildren().add(btn);

            Scene scene = new Scene(root, 400, 300);
            primaryStage.setScene(scene);

            primaryStage.show();
 }
```

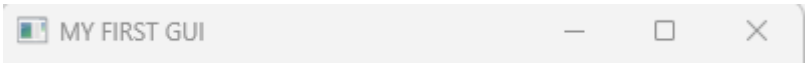
5. Currently our button doesn't do anything and to make our functional button we are going to use event Handler throw adding this code :

```
btn.setOnAction(new EventHandler<ActionEvent>(){
        @Override
        public void handle(ActionEvent event) {
            System.out.println("Hello World!");
        }

   });
```

when the button (btn) is clicked, the code inside the handle method is executed, and "Hello World!" is printed to the console

```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

6. Lastly, we change the title of our GUI using `primaryStage.setTitle("MY FIRST GUI");`



The final code is as follows:

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class JavaFXInterface extends Application {

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

        @Override
        public void start(Stage primaryStage) throws Exception{
            Button btn = new Button("Click here");

            btn.setOnAction(new EventHandler<ActionEvent>(){
                @Override
                public void handle(ActionEvent event) {
                    System.out.println("Hello World!");
                }

            });

            StackPane root = new StackPane();
            root.getChildren().add(btn);

            Scene scene = new Scene(root, 400, 300);
            primaryStage.setScene(scene);
            primaryStage.setTitle("MY FIRST GUI");

            primaryStage.show();
        }

}
```

UI Components:

are elements or building blocks used in graphical user interfaces (GUIs) to allow users to interact with the software or application. UI components are responsible for presenting

information to users and receiving input. In the context of JavaFX or other GUI frameworks, here are some common UI components:

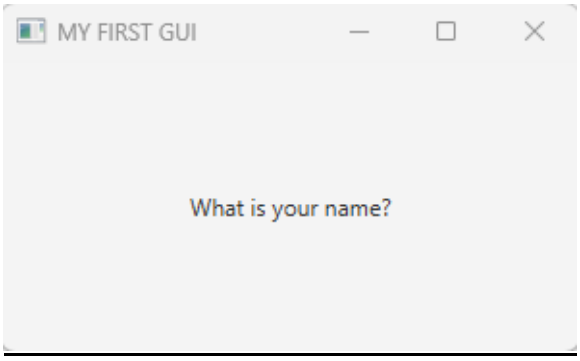
1. **Buttons:** A clickable component that triggers an action when pressed we already talked about them previously.

2. **Labels:** A non-editable text component used to display information or provide instructions. We create a label from the class **Label** found in **javafx.scene.control** Package.

   Import:
   ```
   import javafx.scene.control.Label;
   ```

   declaration:
   ```
   Label label = new Label("Enter your name:");
   ```

   Result :



3. **TextField:** An input component that allows users to enter text.

   Import:
   ```
   import javafx.scene.control.TextField;
   ```

   declaration:
   ```
   TextField textField = new TextField();
   ```

   Result :

4. **ComboBox:**
   A drop-down list that allows users to select an item from a list

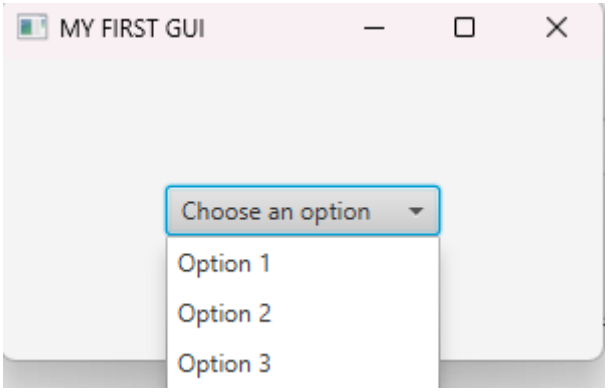
Import:

```
import javafx.scene.control.ComboBox;
```

declaration:

```
// Creating a combobox that will store Strings
ComboBox<String> comboBox = new ComboBox<>();
// Add the different Options in the combo box
comboBox.getItems().addAll("Option 1", "Option 2", "Option 3");
// set the default text prompt
comboBox.setPromptText("Choose an option");
```

Result :



5. **CheckBox:**

A component that represents a binary choice (selected or not selected).

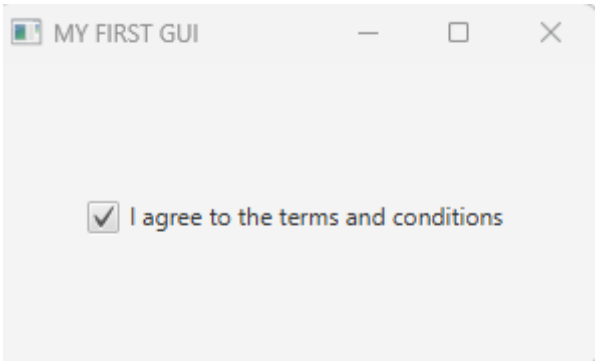
Import:

```
import javafx.scene.control.CheckBox;
```

declaration:

```
CheckBox checkBox = new CheckBox("I agree to the terms and conditions");
```

Result :



6. **RadioButton:**

Similar to a CheckBox but used when only one option from a group can be selected.

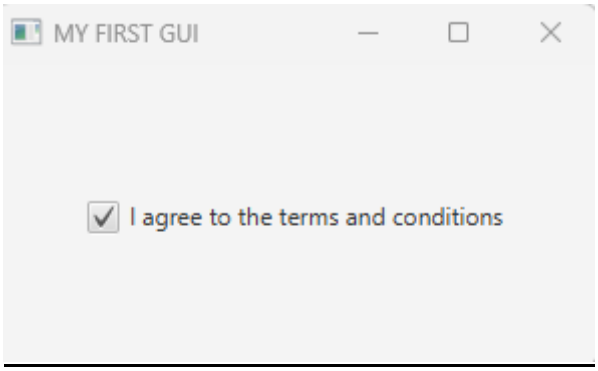
Import:

```
import javafx.scene.control.RadioButton;
```

declaration:

```
RadioButton radioButton = new RadioButton("Option A");
```

Result :



7. **ListView :**

ListView is a powerful JavaFX component that provides a flexible and dynamic way to display lists of items in a graphical user interface.

Import:

```
import javafx.scene.control.ListView;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
```

declaration:

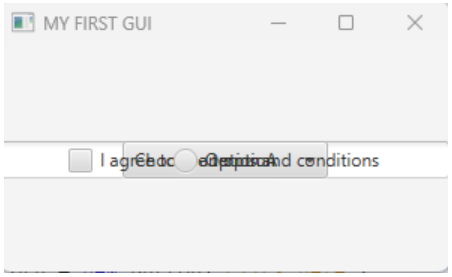
```
ListView<String> listView = new ListView<>();
ObservableList<String> items =
FXCollections.observableArrayList("Item 1", "Item 2", "Item 3");
listView.setItems(items);
```

Result :

There are many other different components that you will discover during your journey of programming with JavaFX and according to your needs.

8. **Adding different components to a layout :**

We discussed earlier how to add a single component each time to our GUI. However, after creating various components, how can we add all of them to the GUI at once? We achieve this using `root.getChildren().addAll(btn, label, textField, comboBox, checkBox, radioButton);` By doing this, we can add all the components to our GUI. However, when you execute your code, you may notice the following issues.



This occurred because we employed StackPane as our layout. To address this issue, we will delve deeper into layouts in the next section.

Layouts:

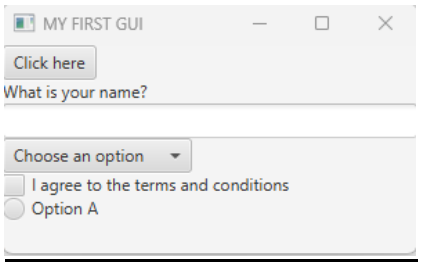
layouts refer to mechanisms for arranging and positioning UI components within a container or a container within another container. Layouts define how components are organized, sized, and aligned, helping to achieve a visually pleasing and well-structured user interface.

1. **StackPane:**
   Components are stacked on top of each other. The last component added is at the top the one that we used previously**.**
2. **HBox and VBox (HorizontalBox and VerticalBox):**
   Components are placed horizontally (HBox) or vertically (VBox) in a single line.

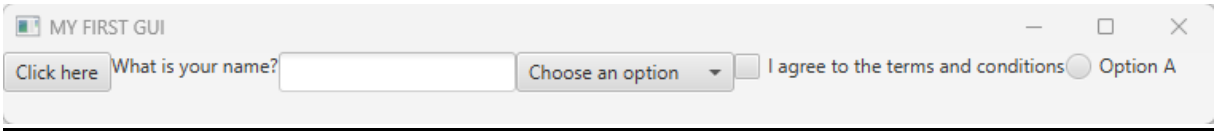
   Import:
   ```
   import javafx.scene.layout.VBox;
   import javafx.scene.layout.HBox;
   ```

   declaration:
   ```
   HBox root = new HBox();
   VBox root = new VBox();
   ```

   Result :



VBox

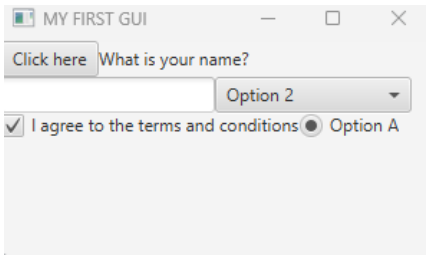


HBox

3. **FlowPane:**
   Components are arranged in a flow, wrapping to the next line when the current line is full.

   Import:
   ```
   import javafx.scene.layout.FlowPane;
   ```

   declaration:
   ```
   FlowPane root = new FlowPane();
   ```
   Result :



4. **GridPane:**
   Components are arranged in a grid, allowing precise control over the placement of each component we will talk more about it in next sessions.

The choice of layout depends on the desired structure of the user interface and the relationships between the components. Different layouts offer different levels of flexibility and control over component positioning.
Layout managers in GUI frameworks automatically handle resizing and repositioning of components based on the chosen layout rules. This allows developers to create responsive and dynamic user interfaces that adapt to changes in window size or content. Overall, layouts play a crucial role in organizing and presenting UI components to enhance the user experience.

Event handlers:

Event handlers define the behavior of UI components in response to user interactions.
Import:
```
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
```

1. **Button Click Event**:
   When a button is clicked, a specific action is triggered.
   Implementation:

```
btn.setOnAction(new EventHandler<ActionEvent>(){
        @Override
         public void handle(ActionEvent event) {
                System.out.println("Hello World!");
        }

 });
```

2. **Mouse Hover Event**:
   When the mouse hovers over a component, a specific action is triggered.

   Import:

```
import javafx.scene.input.MouseEvent;
```

   Implementation:

```
label.setOnMouseEntered(new EventHandler<MouseEvent>() {
        @Override
        public void handle(MouseEvent event) {
        // Define the action to be performed when the mouse enters
        //the label
                System.out.println("Mouse Hovered over the Label!");
        }
});
```

3. **Mouse Click Event**:
   When the mouse clicks on a component, a specific action is triggered.
   Import:

```
import javafx.scene.input.MouseEvent;
```

   Implementation:

```
btn.setOnMouseClicked(new EventHandler<MouseEvent>() {
        @Override
        public void handle(MouseEvent event) {
                // Define the action to be performed when the mouse
                //clicks the Button
                System.out.println("Mouse Clicked on the button!");
         }
});
```

Lambda Expression:

In JavaFX, lambda expressions are often used in event handling to provide concise and expressive ways of defining event handlers. Lambda expressions simplify the syntax for defining single-method interfaces, such as the EventHandler interface in JavaFX, which is commonly used to handle events like button clicks, mouse events, and other user interactions.

Here's a brief explanation of how lambda expressions work in JavaFX:

1. **Traditional Event Handling:**
   Before lambda expressions were introduced in Java, event handling in JavaFX was done using anonymous inner classes. For example, handling a button click event looked like this:

```
Button button = new Button("Click me");
button.setOnAction(new EventHandler<ActionEvent>() {
```

```
@Override
public void handle(ActionEvent event) {
        System.out.println("Button clicked!");
}
    });
```

2. **Using Lambda Expressions:**
   With lambda expressions, the same event handling can be achieved with a more concise syntax:

```
Button button = new Button("Click me");
button.setOnAction(event -> System.out.println("Button clicked!"));
```

Event handlers play a crucial role in JavaFX applications by allowing developers to respond to user interactions and events effectively.

Value Retrieval from UI Components (Getters)

Value retrieval from UI components is a fundamental aspect of JavaFX programming that allows developers to obtain the current state or content of various user interface elements. This process involves using getter methods provided by JavaFX components to access information such as text input, selected options, or other states. Effective value retrieval is crucial for processing user input, making dynamic decisions, and maintaining synchronization between the application's logic and its visual representation. Here is a list of different retrievals in different components

1. **In TextField**:

```
TextField nameTextField = new TextField();
String enteredName = nameTextField.getText();
```

   In this example, the getText() method retrieves the text entered by the user in a TextField component, allowing developers to capture and process textual input.

2. **In ComboBox:**

```
ComboBox<String> genderComboBox = new ComboBox<>();
String selectedGender = genderComboBox.getValue();
```

   The getValue() method is used to retrieve the currently selected item in a ComboBox. This is valuable for obtaining user selections from a list of options.

3. **In RadioButton in a ToggleGroup:**

```
ToggleGroup group = new ToggleGroup();
RadioButton optionARadioButton = new RadioButton("Option A");
optionARadioButton.setToggleGroup(group);
RadioButton selectedRadioButton = (RadioButton)
group.getSelectedToggle();
```

Retrieving the selected RadioButton from a ToggleGroup involves using the getSelectedToggle() method, ensuring that only one option can be selected at a time within the group.

4. **In CheckBox:**

```
CheckBox agreementCheckBox = new CheckBox("I agree to the terms");
boolean isAgreed = agreementCheckBox.isSelected();
```

The isSelected() method retrieves the current state of a CheckBox, indicating whether it is checked or unchecked. This is useful for capturing binary choices in the user interface.

Mastering the art of value retrieval from UI components in JavaFX is essential for creating responsive and user-centric applications. The ability to extract meaningful information from different components empowers developers to make informed decisions based on user input, enhancing the overall user experience and functionality of the application. As we delve into specific examples, understanding these retrieval mechanisms becomes paramount for effective JavaFX development.

Value Setting (Setters)

Value setting, facilitated by the use of setter methods in JavaFX, is a pivotal aspect of application development. This process empowers developers to update and customize the state, appearance, or behavior of various UI components dynamically. By employing setter methods tailored to each JavaFX component, developers can effectively inject values, modify text content, or configure other properties. This is fundamental for creating interactive and adaptive user interfaces, allowing applications to respond seamlessly to changing conditions or user input. In the upcoming examples, we explore how to set values for different components in JavaFX. Value Setting Examples:

1. **For Labels**:

```
Label nameLabel = new Label();
nameLabel.setText("User Name:");
```

The setText() method is used to set the text content of a Label, providing information or instructions to the user.

2. **For ObservableList**:

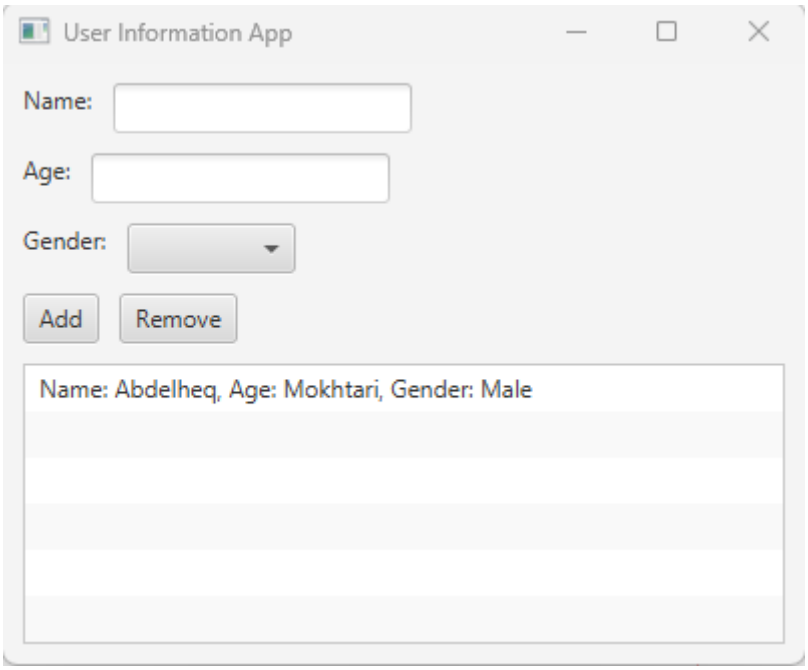
```
ListView<String> stringListView = new ListView<>();
ObservableList<String> items =
FXCollections.observableArrayList("Item 1", "Item 2", "Item 3");
stringListView.setItems(items);
```

The setItems() method is used to set the observable list of items that the ListView will display. This is crucial for populating the list with meaningful data.

Value setting is a fundamental skill in JavaFX development, offering developers the ability to dynamically configure and customize UI components. Through the use of setter methods, developers can tailor the appearance and behavior of their applications, creating adaptive and user-friendly interfaces. As we delve into specific examples of value setting, a deeper understanding of these mechanisms becomes essential for effective JavaFX programming.

Application

An example using JavaFX with the use of what we had learn so far



Source code :

```
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.geometry.Insets;
```

```
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.stage.Stage;

public class UserInformationApp extends Application {

    private ObservableList<String> personList =
FXCollections.observableArrayList();

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

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("User Information App");

        // Create UI components
        TextField nameField = new TextField();
        TextField ageField = new TextField();
        ComboBox<String> genderComboBox = new
ComboBox<>(FXCollections.observableArrayList("Male", "Female"));
        Button addButton = new Button("Add");
        Button removeButton = new Button("Remove");
        ListView<String> personListView = new ListView<>(personList);

        // Set button actions
        addButton.setOnAction(event -> {
            String name = nameField.getText();
            String age = ageField.getText();
            String gender = genderComboBox.getValue();

            if (!name.isEmpty() && !age.isEmpty() && gender != null) {
                String personInfo = "Name: " + name + ", Age: " + age + ",
Gender: " + gender;
                personList.add(personInfo);

                // Clear input fields
                nameField.clear();
                ageField.clear();
                genderComboBox.getSelectionModel().clearSelection();
            } else {
                showAlert("Incomplete Information", "Please fill in all
fields.");
            }
        });

        removeButton.setOnAction(event -> {
            int selectedIndex =
personListView.getSelectionModel().getSelectedIndex();
            if (selectedIndex != -1) {
                personList.remove(selectedIndex);
            } else {
                showAlert("No Selection", "Please select a person to
remove.");
            }
        });

        // Create layout
        VBox root = new VBox(10);
```

```
        root.setPadding(new Insets(10));
        root.getChildren().addAll(
                new HBox(10, new Label("Name:"), nameField),
                new HBox(10, new Label("Age:"), ageField),
                new HBox(10, new Label("Gender:"), genderComboBox),
                new HBox(10, addButton, removeButton),
                personListView
        );

        // Set up the scene
        Scene scene = new Scene(root, 400, 300);
        primaryStage.setScene(scene);

        // Show the stage
        primaryStage.show();
    }

    private void showAlert(String title, String content) {
        Alert alert = new Alert(Alert.AlertType.INFORMATION);
        alert.setTitle(title);
        alert.setHeaderText(null);
        alert.setContentText(content);
        alert.showAndWait();
    }
}
```

Try to copy and execute the code understand each instruction and try to build your own version based on this code