Part 1

CMPS 251

Read Chapter 12

JavaFX™

Graphical User Interfaces (GUI)

Dr. Abdelkarim Erradi

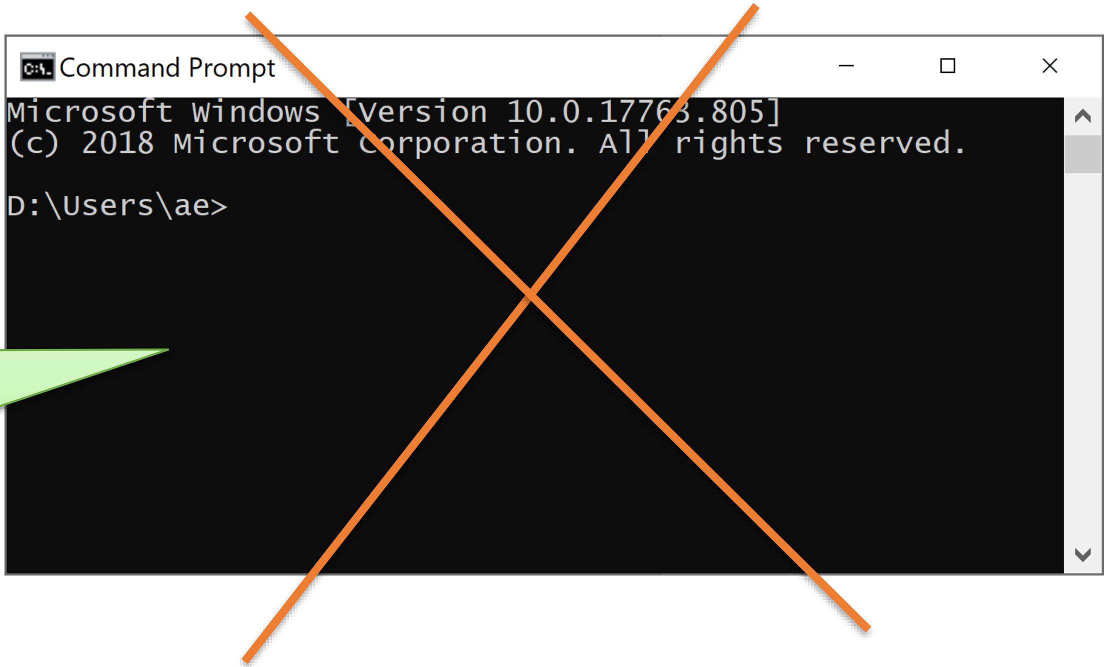
CSE@QU

Outline

- GUI Programming Model
- Model-View-Controller (MVC) Pattern
- JavaFX Layout
- Handling Events

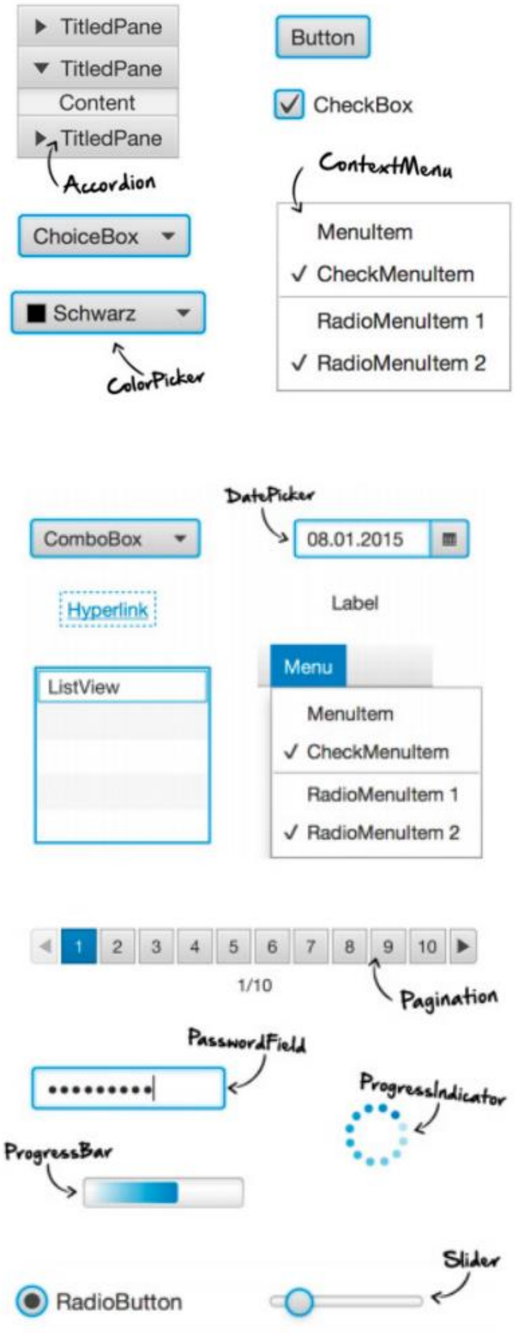
GUI Programming Model

You have open holidays!
We might send you
to the **Museum** ☺



```
Microsoft Windows [Version 10.0.17763.805]
(c) 2018 Microsoft Corporation. All rights reserved.

D:\Users\ae>
```

< Back

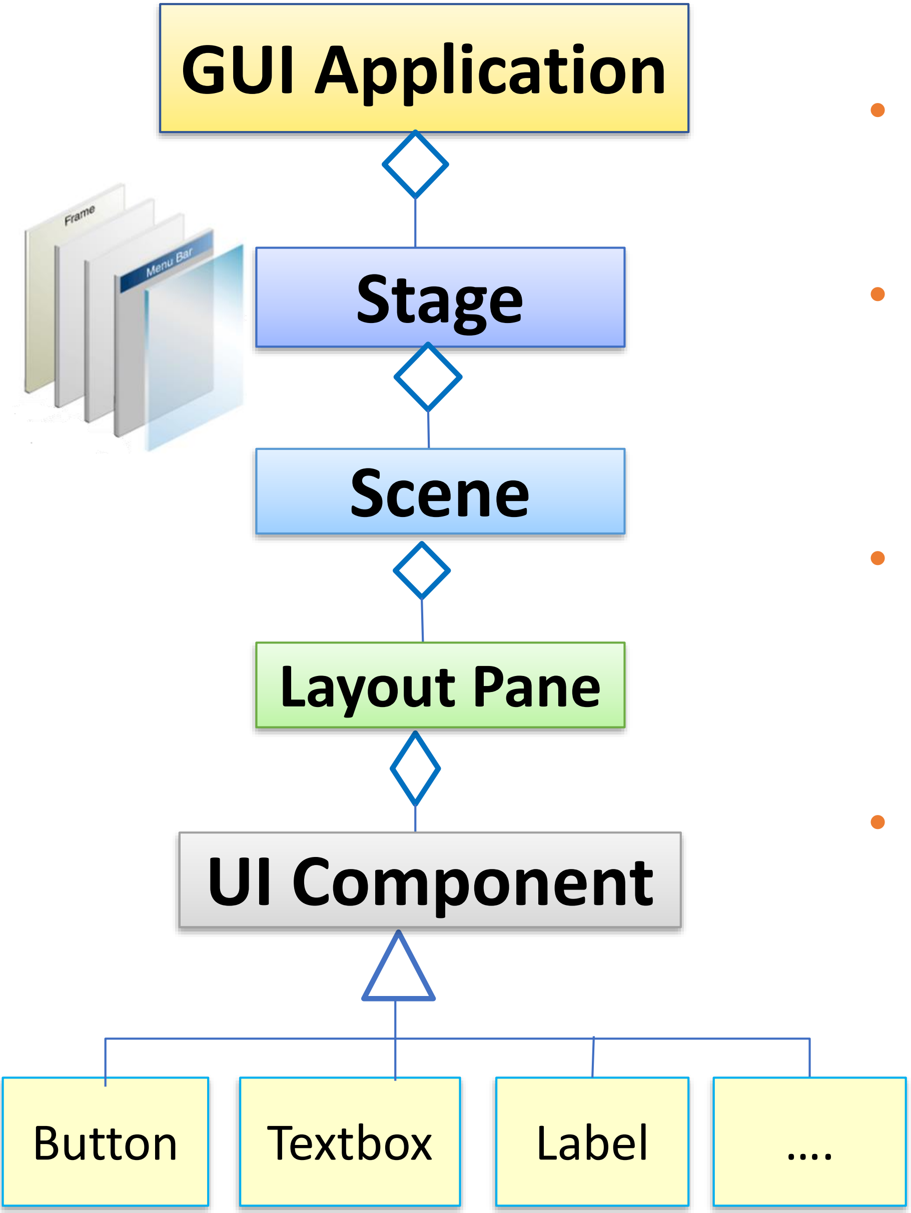


What is a GUI?

- **Graphical User Interface (GUI)** provides a visual User Interface (واجهة الاستخدام) for the users to interact with the application
  - ○ Instead of a Character-based interface provided by the console interface ‘the scary black screen’ >_
- **JavaFX** can be used for creating GUI

GUI Programming Model

IMPORTANT

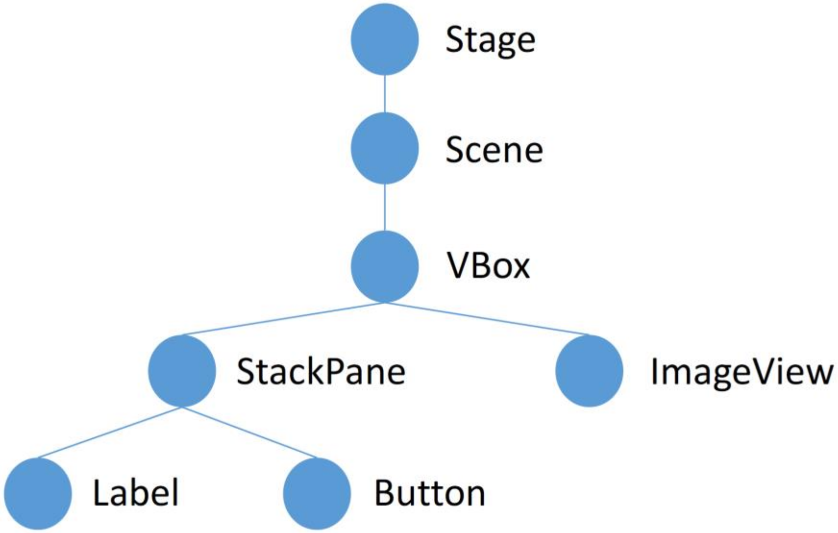


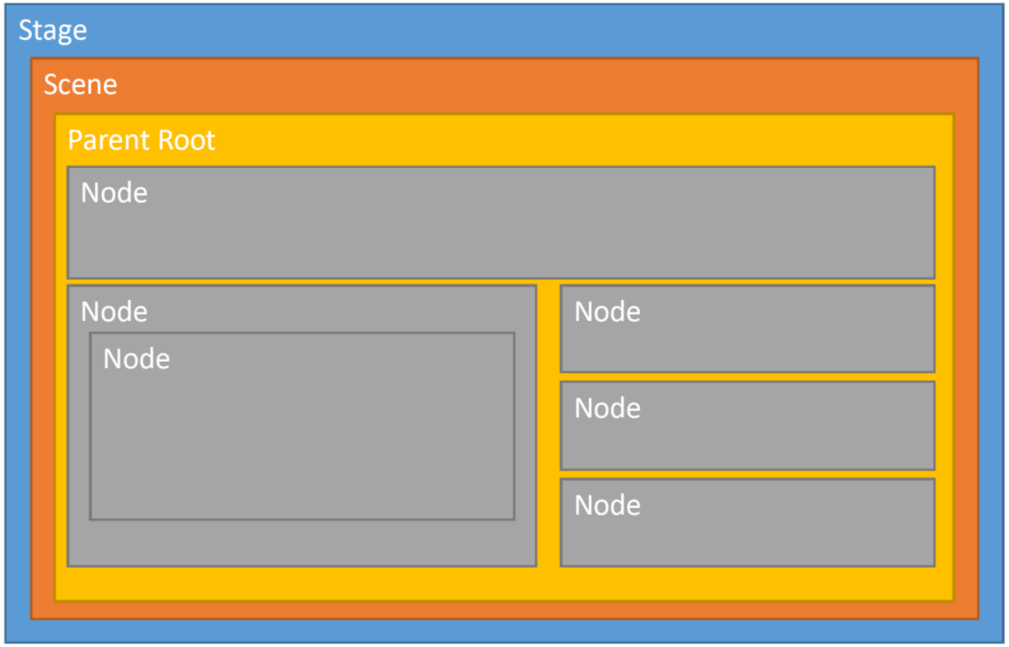
- GUI of an application is made up of **Windows** (JavaFX calls it **Stage**)
- A window has a **container** (called **Scene**) to host the UI root layout container
- UI Components are first added to a root layout container (such as VBox) then placed in the Scene
- UI Components **raise Events** when the user interacts with them (such as a MouseClicked event is raised when a button is clicked).
  - Programmer write **Event Handlers** to respond to the UI events

Structure of JavaFX application

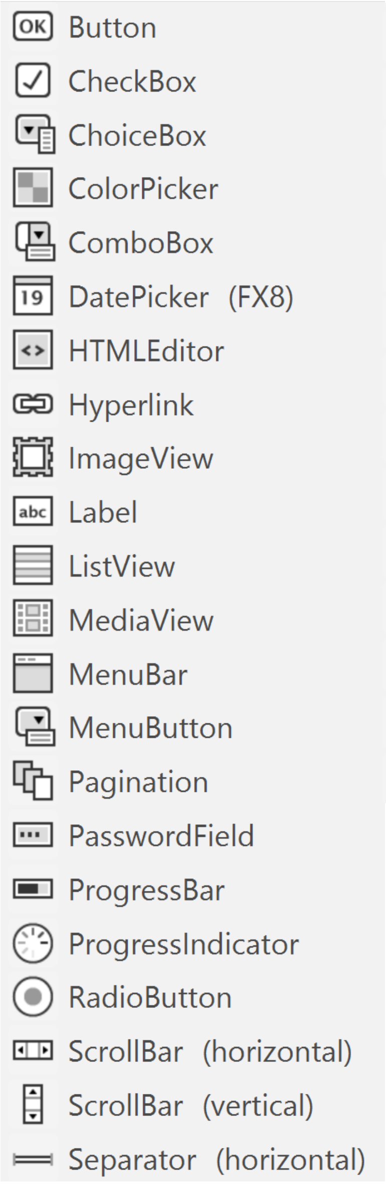
Stage = **Window** where a scene is displayed

Scene = **Container** to host the UI root layout container

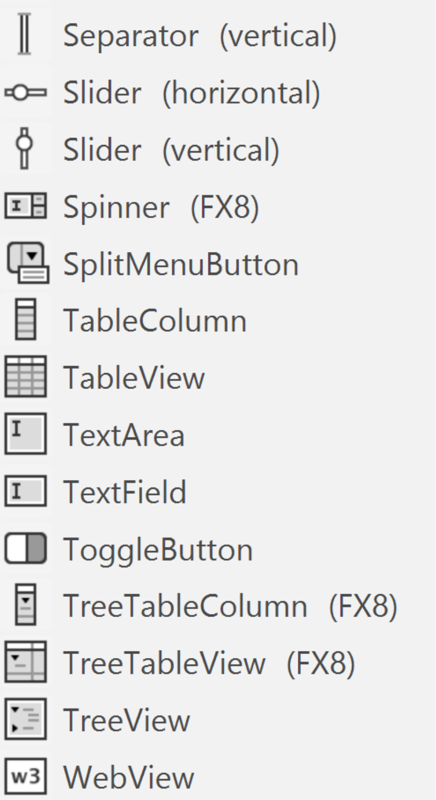




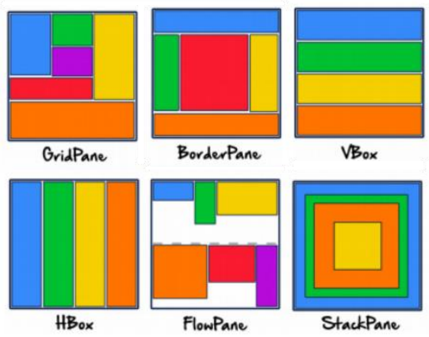
What Makes up JavaFx ?



Button
CheckBox
ChoiceBox
ColorPicker
ComboBox
DatePicker (FX8)
HTMLEditor
Hyperlink
ImageView
Label
ListView
MediaView
MenuBar
MenuButton
Pagination
PasswordField
ProgressBar
ProgressIndicator
RadioButton
ScrollBar (horizontal)
ScrollBar (vertical)
Separator (horizontal)

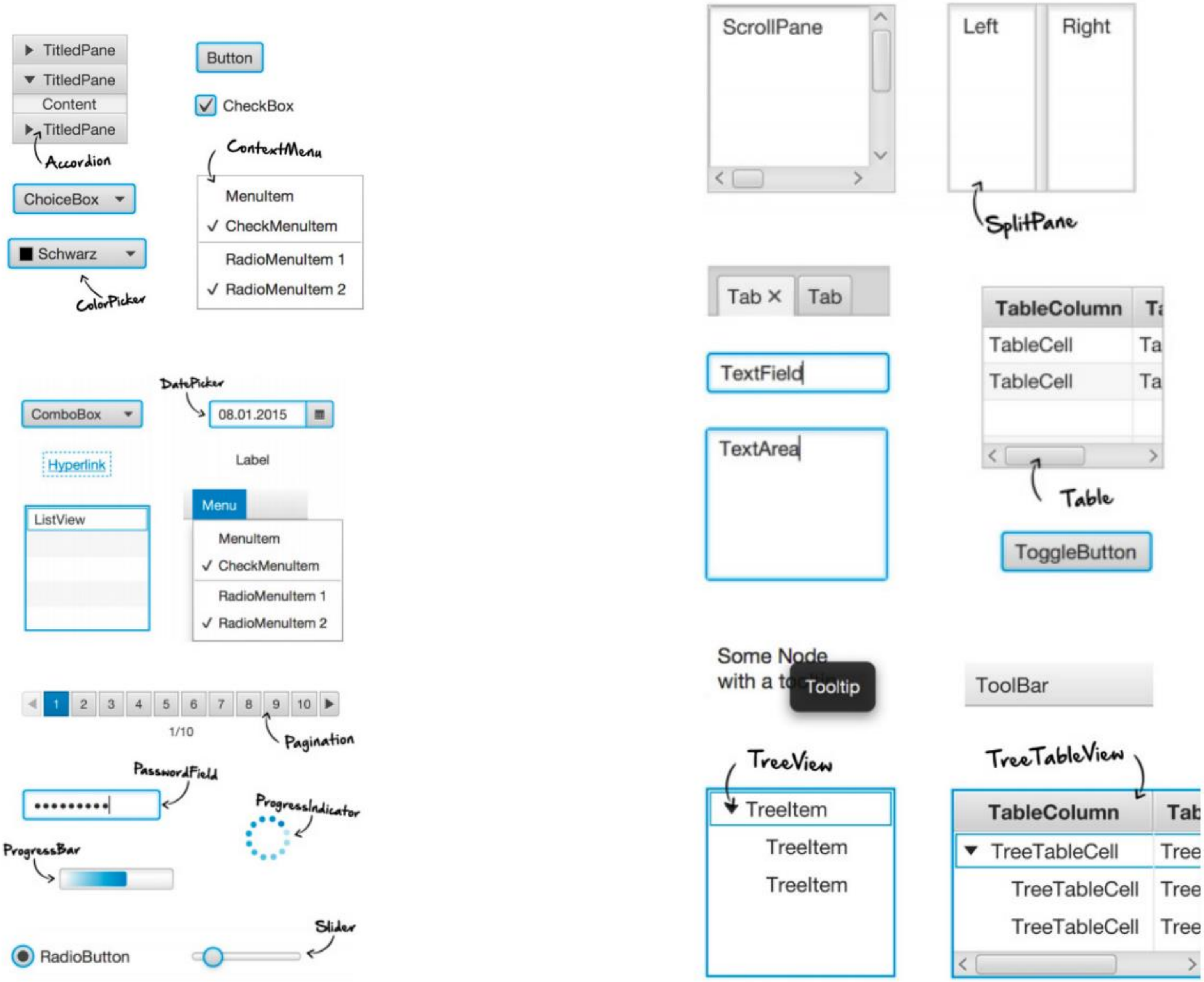


Separator (vertical)
Slider (horizontal)
Slider (vertical)
Spinner (FX8)
SplitMenuButton
TableColumn
TableView
TextArea
TextField
ToggleButton
TreeTableColumn (FX8)
TreeTableView (FX8)
TreeView
WebView

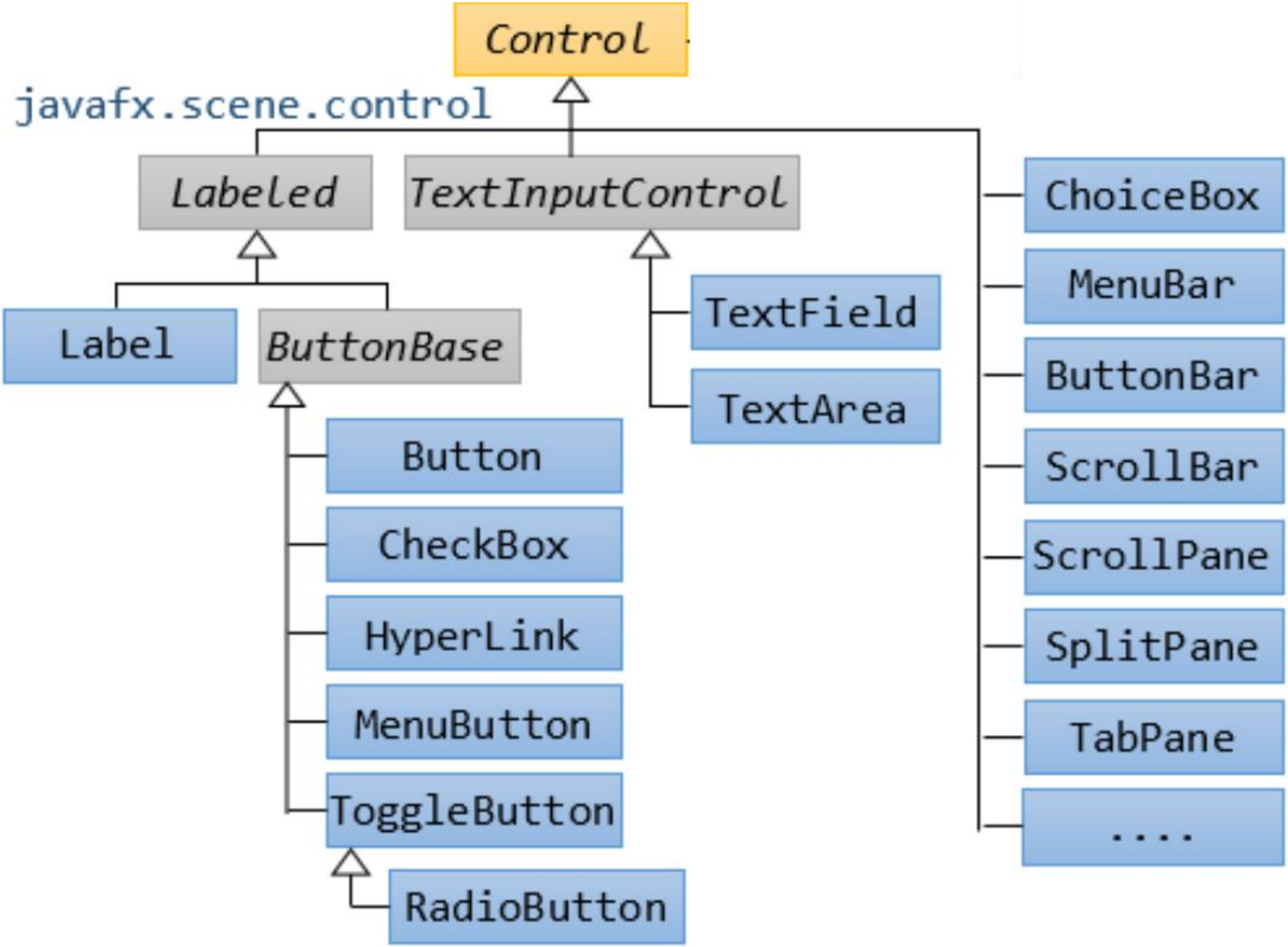


- **UI components**
  - Set of pre-built UI components that can be composed to create a GUI
  - e.g. buttons, text-fields, menus, tables, lists, etc.
- **Layout containers**
  - Control placement/ positioning of components in the form (e.g., VBox and HBox)

JavaFX UI Components



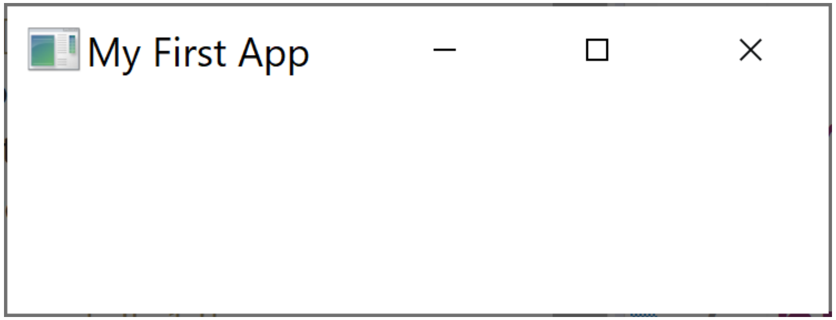
JavaFX UI Components Hierarchy



Creating JavaFX GUI: Stage (1/2)

- Create a class that extends `javafx.application.Application`
- Implement the `start(Stage stage)` method to build and display the UI
  - `start()` is called when the app is launched
- JavaFX automatically creates an instance of **`Stage`** class and passes to `start()`
  - when `start()` calls `stage.show()` a window is displayed

```java
public class App extends Application {
 @Override
 public void start(Stage stage) {
   stage.setTitle("My First App");
   stage.show();
 }

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



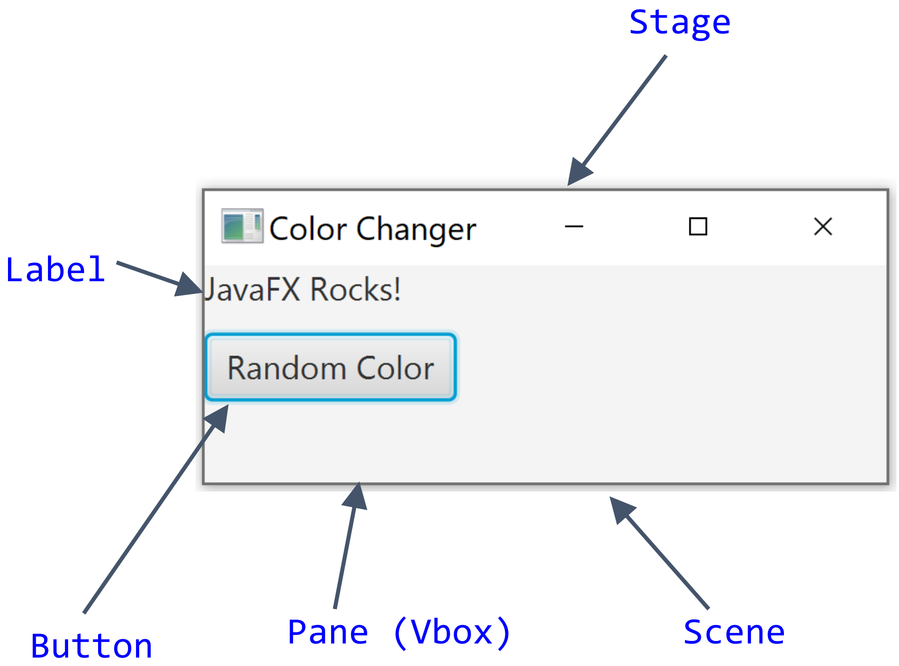
Creating JavaFX GUI : Scene (2/2)

- Create a **scene** (instance of `javafx.scene.Scene`) within the `start` method as the top-level container for the UI components
    - then pass the `scene` to the `stage` using the `setScene` method
- UI components (a Button, a Label…) can be added to a layout container (e.g., VBox) then added to the `Scene` to get displayed

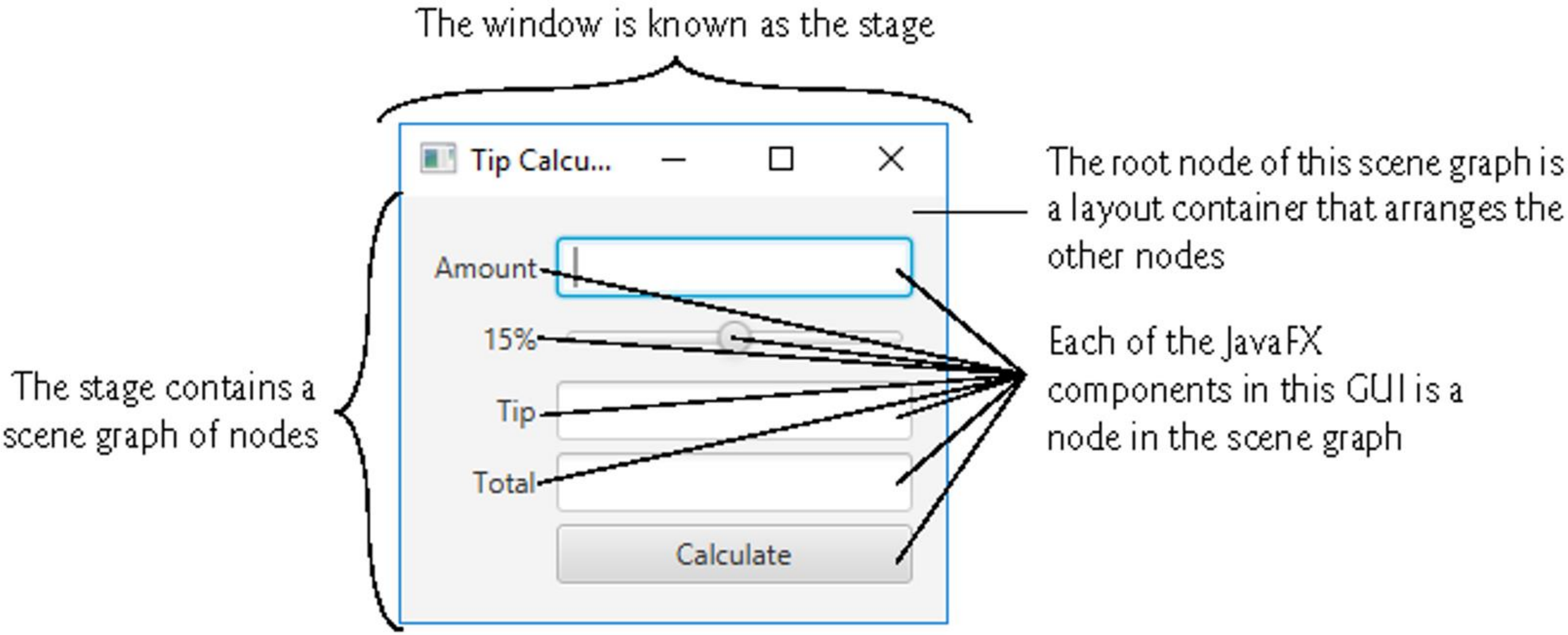
```java
public void start(Stage stage) {
   VBox root = new VBox();
   Label label = new Label("JavaFX Rocks!");
   Button button = new Button("Submit");
   root.getChildren().addAll(label, button);
   Scene scene = new Scene(root, 200, 200);
   stage.setScene(scene);
   stage.show();
}
```

JavaFX Application: ColorChanger

- App that contains text reading “JavaFX Rocks!” and a Button that randomly changes text’s color with every click



JavaFX App Components

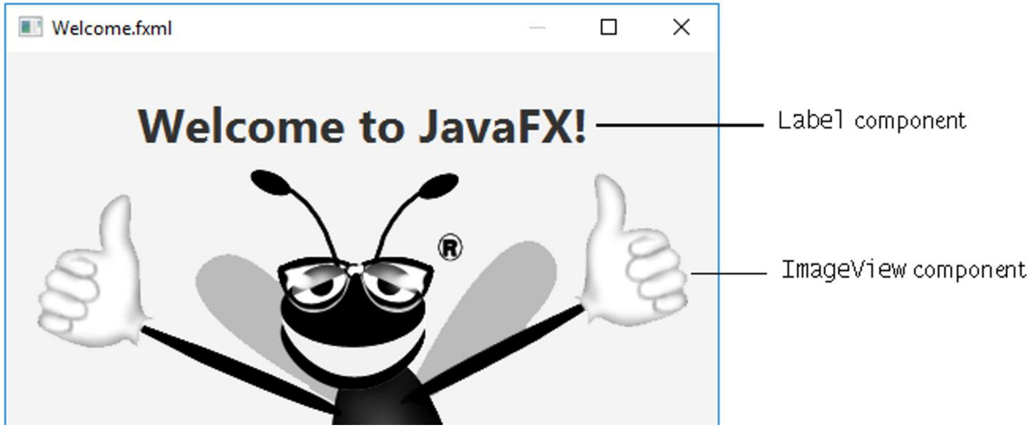


The window is known as the stage

The root node of this scene graph is a layout container that arranges the other nodes

Each of the JavaFX components in this GUI is a node in the scene graph

The stage contains a scene graph of nodes

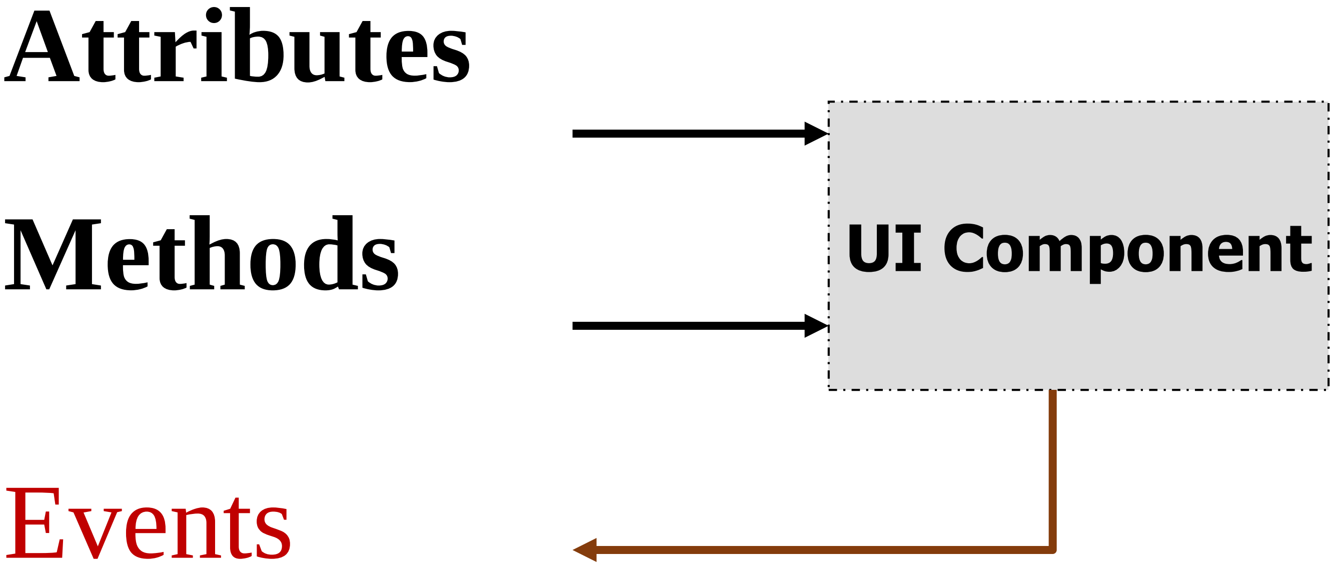


Label component

ImageView component

UI Component

- UI component is a class that has:



Attributes

Methods

Events

UI Component

Using a UI Component

1. **Create it**

```
Button button = new Button("SUbmit");
```

2. **Initialize it / configure it**

```
button.setTextFill( Color.BLUE );
```

3. **Add it to a layout container**

```
vBox.add(button);
```

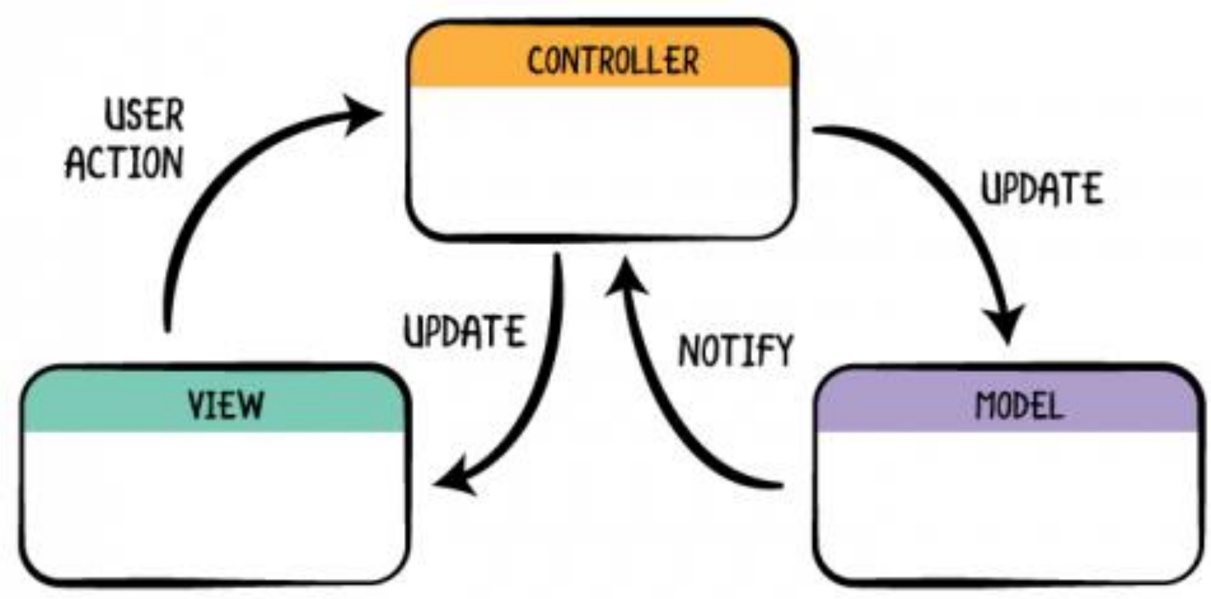
Steps 1 to 3 can be done using **Scene Builder**

4. **Listen to and handle its events**

```
// Register an event handler
button.setOnActionEvent( event ->
        System.out.println(event) );
```

IMPORTANT

Model-View-Controller (MVC) Pattern



< Back

MVC = decompose the app into 3 parts: Model, View and Controller

IMPORTANT

View

- Gets input from the user
- Notifies the controller about UI events
- Displays output to the user

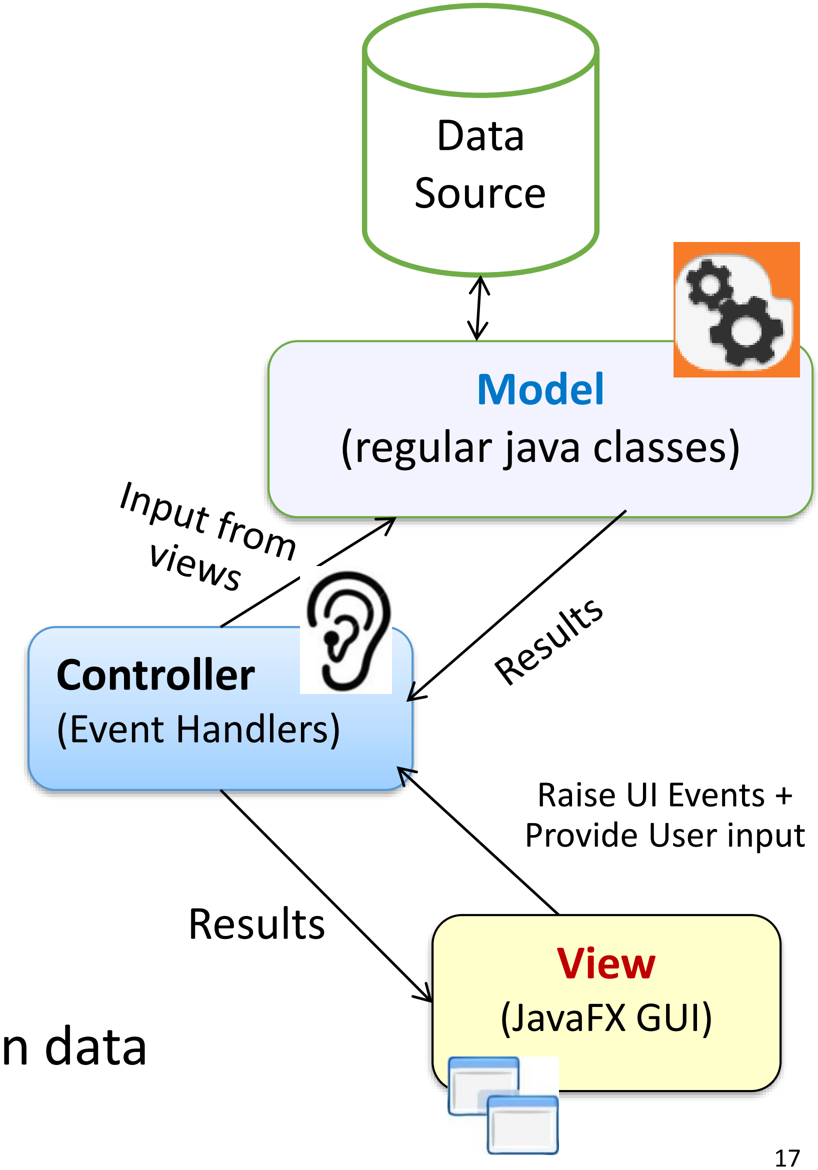
Controller

- Handles events raised by the view
- Instructs the model to perform actions based on user input

e.g. request the model to get the list of courses

- Passes the results to the view to display the output

Model – implements business logic and computation, and manages the application data



Implementing MVC with JavaFX

1. Define the **model** (*Java classes*) to represent data and encapsulate computation

2. Build the **view** (using Scene Builder tor code) to collect input from the user and displays the results received from the controller

3. Use a **controller** (Java class) to listen to and **handle events** raised by the view

   - Controller coordinates the execution of the request, get the request parameters from the View, calls the model to obtain the results (i.e., objects from the model)

   - Pass the results to the view to display the output

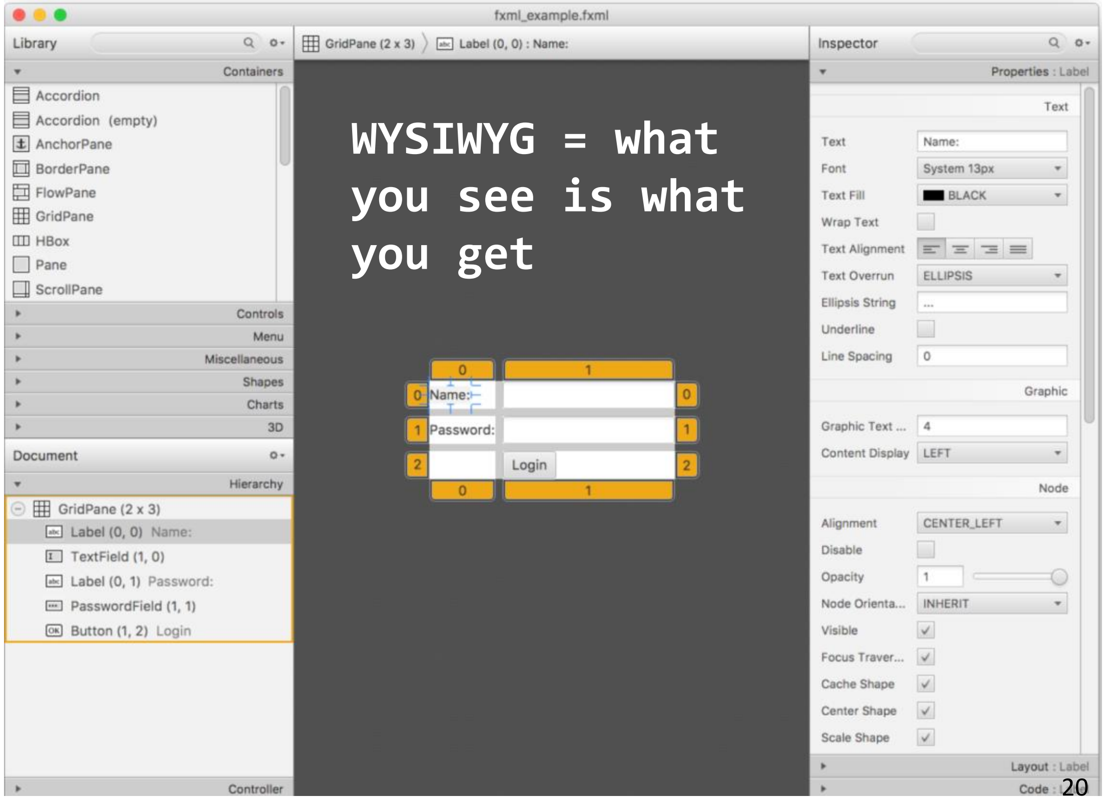
Advantages of MVC

- ***Separation of concerns***
  - Views, controller, and model are **separate components**
    - Computation is not intermixed with Presentation. Consequently, code is cleaner, flexible and easier to understand and change.
    - Allow changing a component without significantly disturbing the others (e.g., UI can be completely changed without touching the model)
- **Reusability**
  - The same model can used by different views (e.g., JavaFX view, Web view and Mobile view)

MVC is widely used and recommended particularly for interactive applications with GUI

Building the View using FXML

- You can create the View using Java code or FXML
- FXML is an XML-based language that defines the **structure** and **layout** of the View
- FXML allows a **clear separation** between the view of an app and the logic
- SceneBuilder is a WYSIWYG editor for FXML



WYSIWYG = what you see is what you get

fxml_example.fxml

GridPane (2 x 3) 〉 Label (0, 0) : Name:

Hierarchy

- GridPane (2 x 3)
  - Label (0, 0) Name:
  - TextField (1, 0)
  - Label (0, 1) Password:
  - PasswordField (1, 1)
  - Button (1, 2) Login

Properties : Label

Text

Text: Name:
Font: System 13px
Text Fill: BLACK
Wrap Text: □
Text Overrun: ELLIPSIS
Ellipsis String: ...
Underline: □
Line Spacing: 0

Graphic

Graphic Text ...: 4
Content Display: LEFT

Node

Alignment: CENTER_LEFT
Disable: □
Opacity: 1
Node Orienta...: INHERIT
Visible: ☑
Focus Traver...: ☑
Cache Shape: ☑
Center Shape: ☑
Scale Shape: ☑

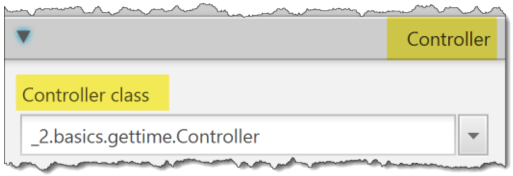
Loading FXML file into a stage

```
@Override
public void start(Stage stage) throws Exception {
   //Parent is a base class for all nodes that have children
   Parent root =
    FXMLLoader.load(getClass().getResource("welcome.fxml"));
   stage.setTitle("Welcome to JavaFX");
   stage.setScene(new Scene(root, 400, 300));
   stage.show();
}
```

FXML Controller

- FXML file is associated with a **Controller** class that implements the events handlers
  - Controller class name must be assigned to **`fx:controller`** attribute of the FXML view
- The Controller defines:
  - **attributes** annotated with **@FXML** to refer to UI elements *to be* accessed programmatically
    - Attribute name defined in the controller must be exactly the same as the UI component name assigned to **`fx:id`** using SceneBuilder
  - ***event handlers*** annotated with **@FXML**
    - Event handler name defined in the controller must be exactly the same as the event handler name assigned the corresponding UI element using SceneBuilder

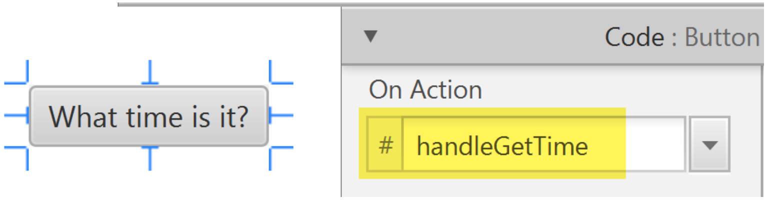
FXML + Controller



Controller class

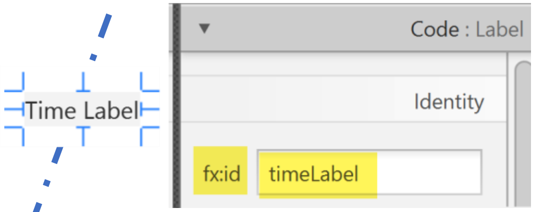
_2.basics.gettime.Controller

```
<VBox fx:controller="gettime.Controller">
    <children>
        <Label text="Time Label" fx:id="timeLabel" />
        <Button text="What time is it?"
                onAction="#handleGetTime" />
    </children>
</VBox>
```



On Action

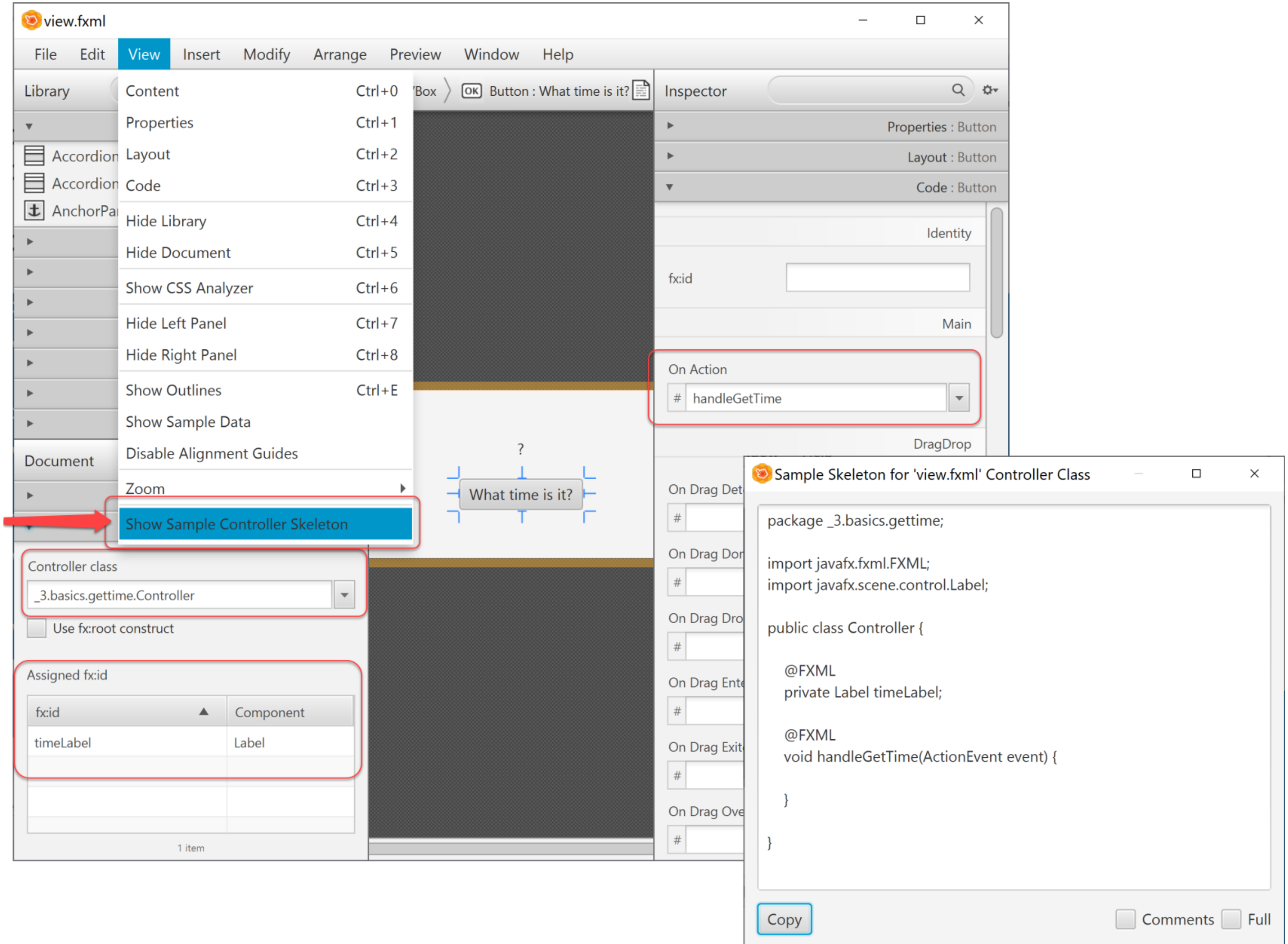
# handleGetTime



fx:id timeLabel

```
public class Controller {
    @FXML private Label timeLabel;

    @FXML void handleGetTime(ActionEvent event) {
    timeLabel.setText(Model.getTime());
    }
}
```

Once you set the fx:id of UI elements and Event Handlers in the FXML you can generate a skeleton Controller class



Controller class

_3.basics.gettime.Controller

□ Use fx:root construct

Assigned fx:id

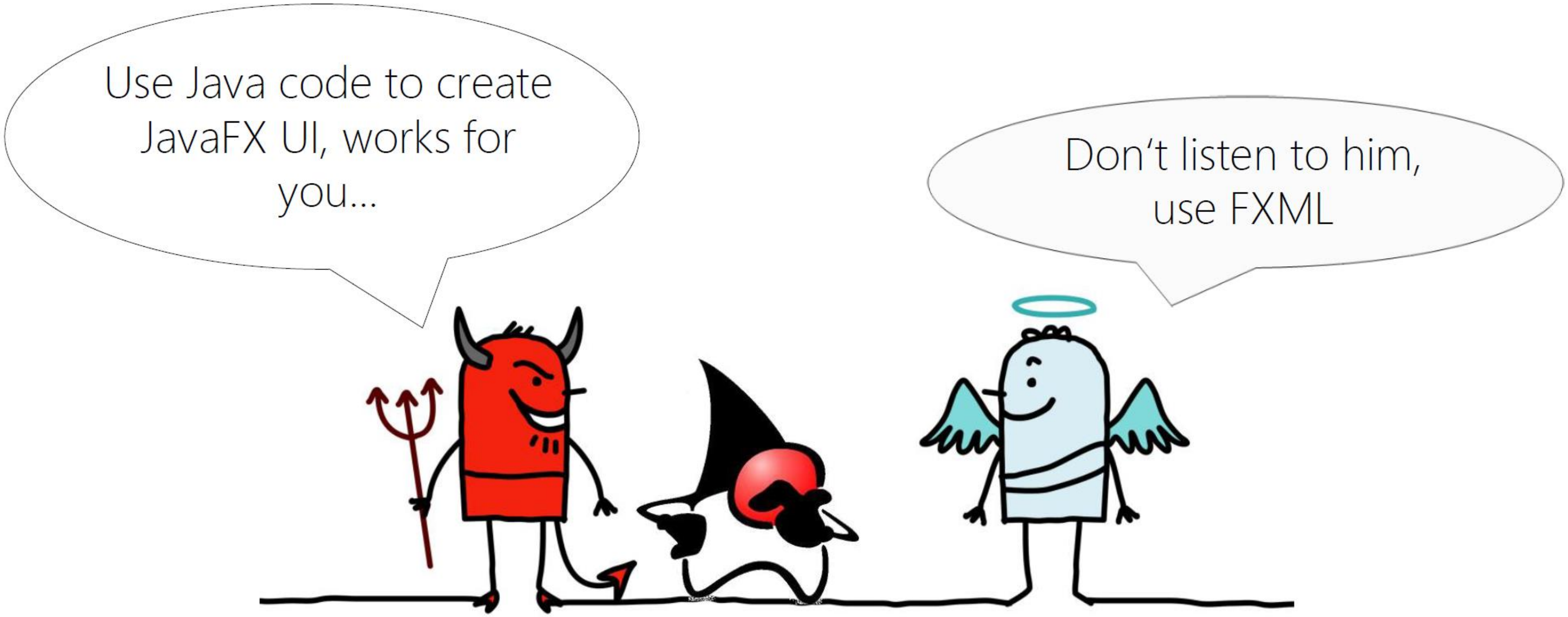
fx:id ▲	Component
timeLabel	Label

1 item

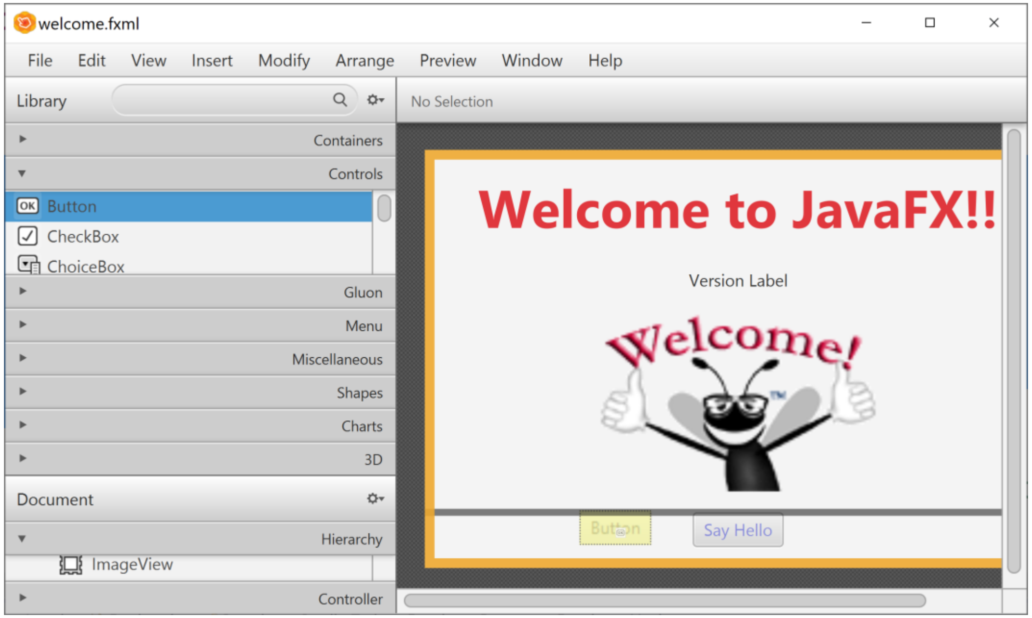
On Action

# handleGetTime

```
package _3.basics.gettime;

import javafx.fxml.FXML;
import javafx.scene.control.Label;

public class Controller {

    @FXML
    private Label timeLabel;

    @FXML
    void handleGetTime(ActionEvent event) {

    }

}
```


Use Java code to create JavaFX UI, works for you...
Don't listen to him, use FXML

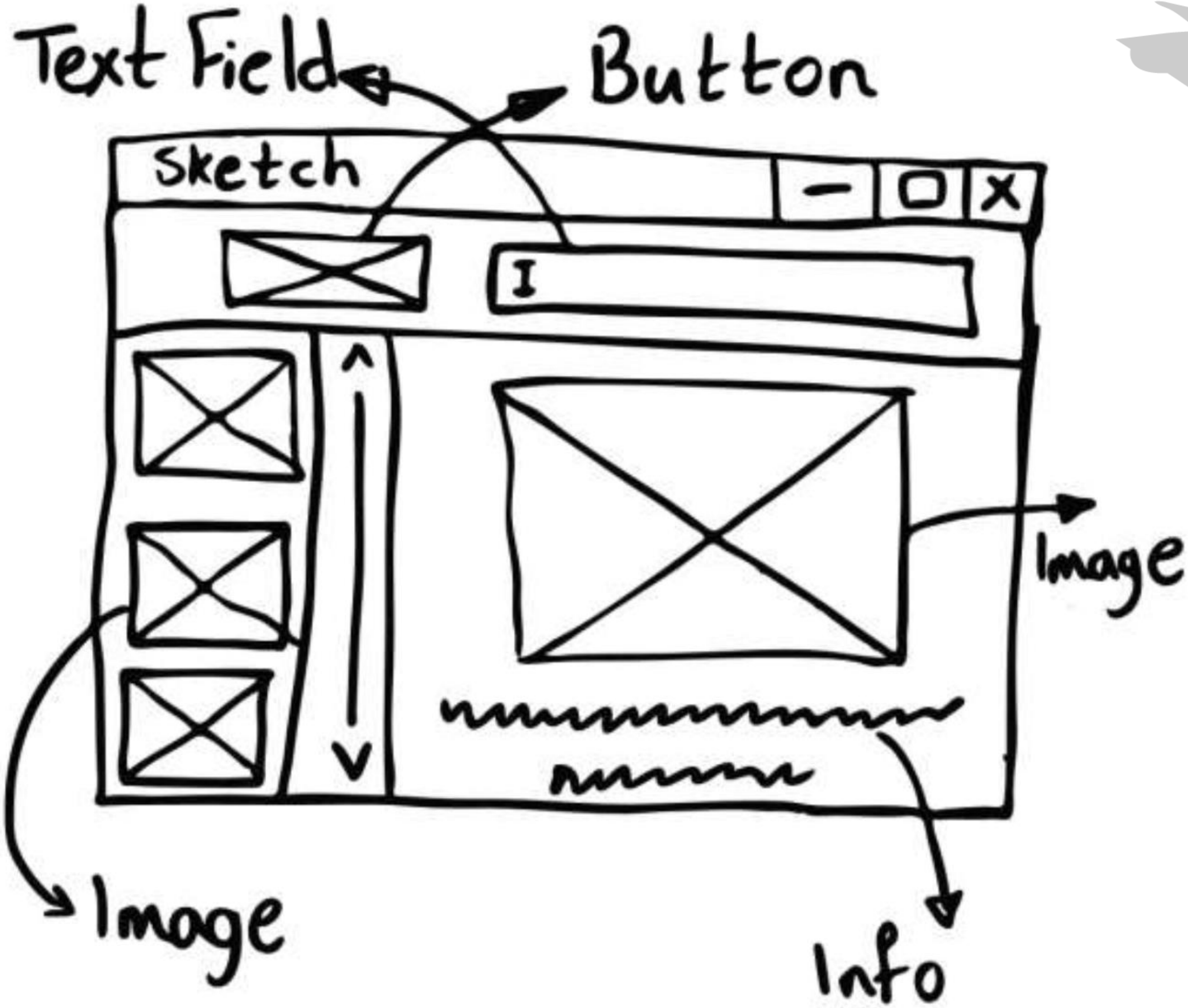
```
VBox root = new VBox();
Label label = new Label("JavaFX Rocks!");
Button button = new Button("Random Color");
button.setTextFill(Color.BLUE);
root.getChildren().addAll(label, button);
root.setSpacing(20);
root.setAlignment(Pos.CENTER);
```

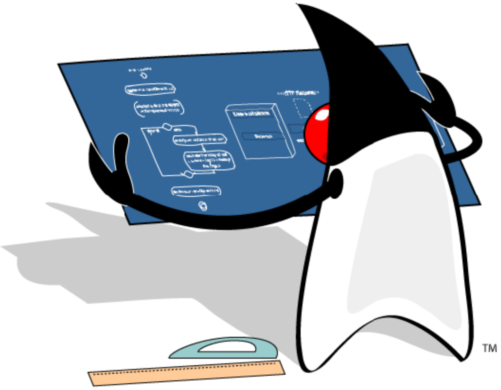


Steps to creating a GUI Interface

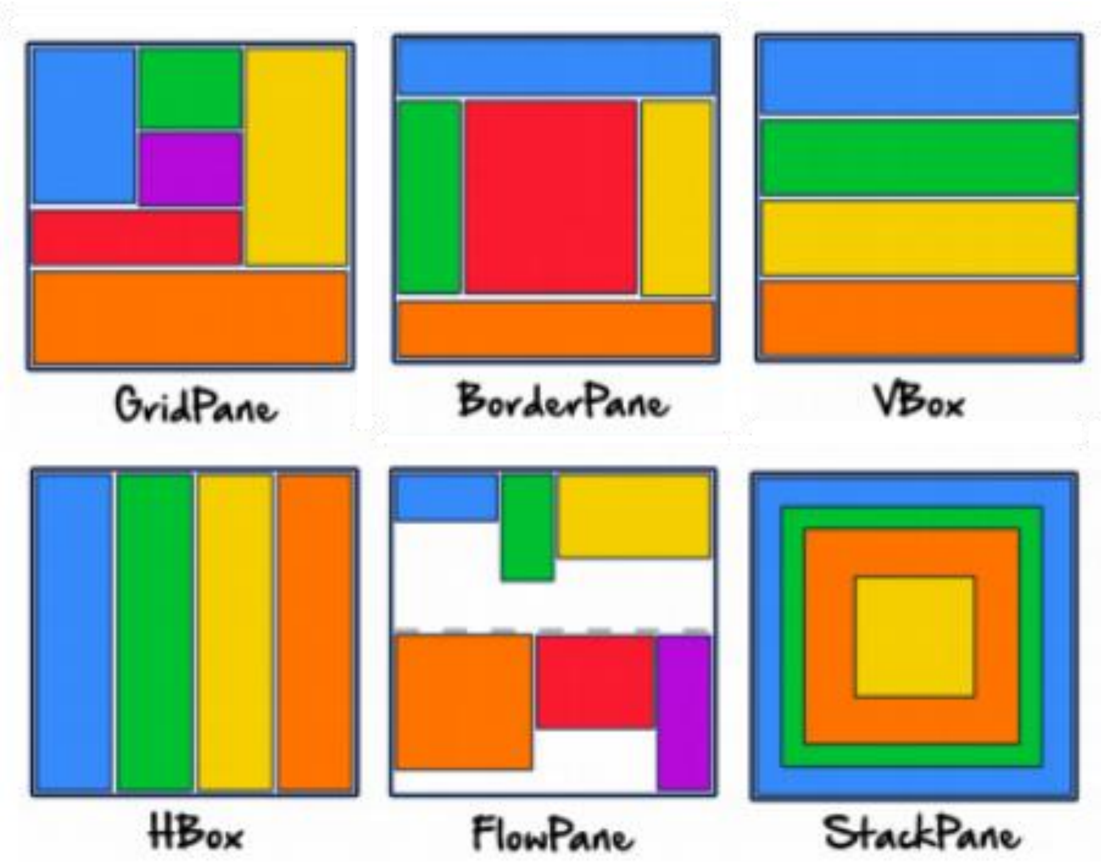
1. Design it on paper (sketch)
   - Decide what information to present to user and what input they should supply
   - Decide the UI components and the layout on paper
2. Create a view and add components to it (using either SceneBuilder or java code)
   - Use layout panes to group and arrange components
3. Add event handlers to respond to the user actions (event driven programming)
   - Do something when the user presses a button, moves the mouse, change text of input field, etc.

UI Sketch - Example





Layouts



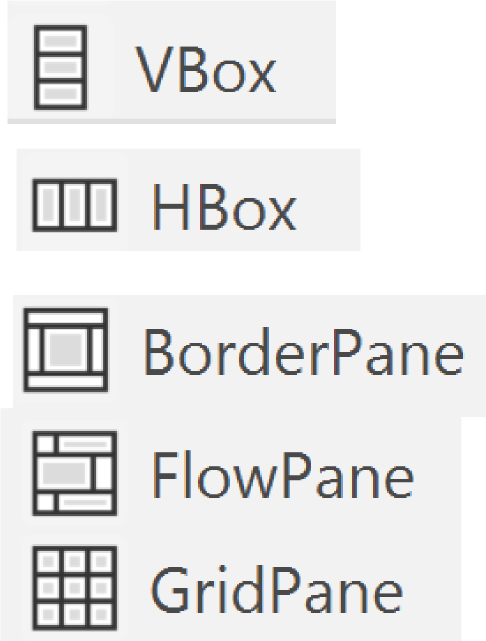
< Back

Layouts

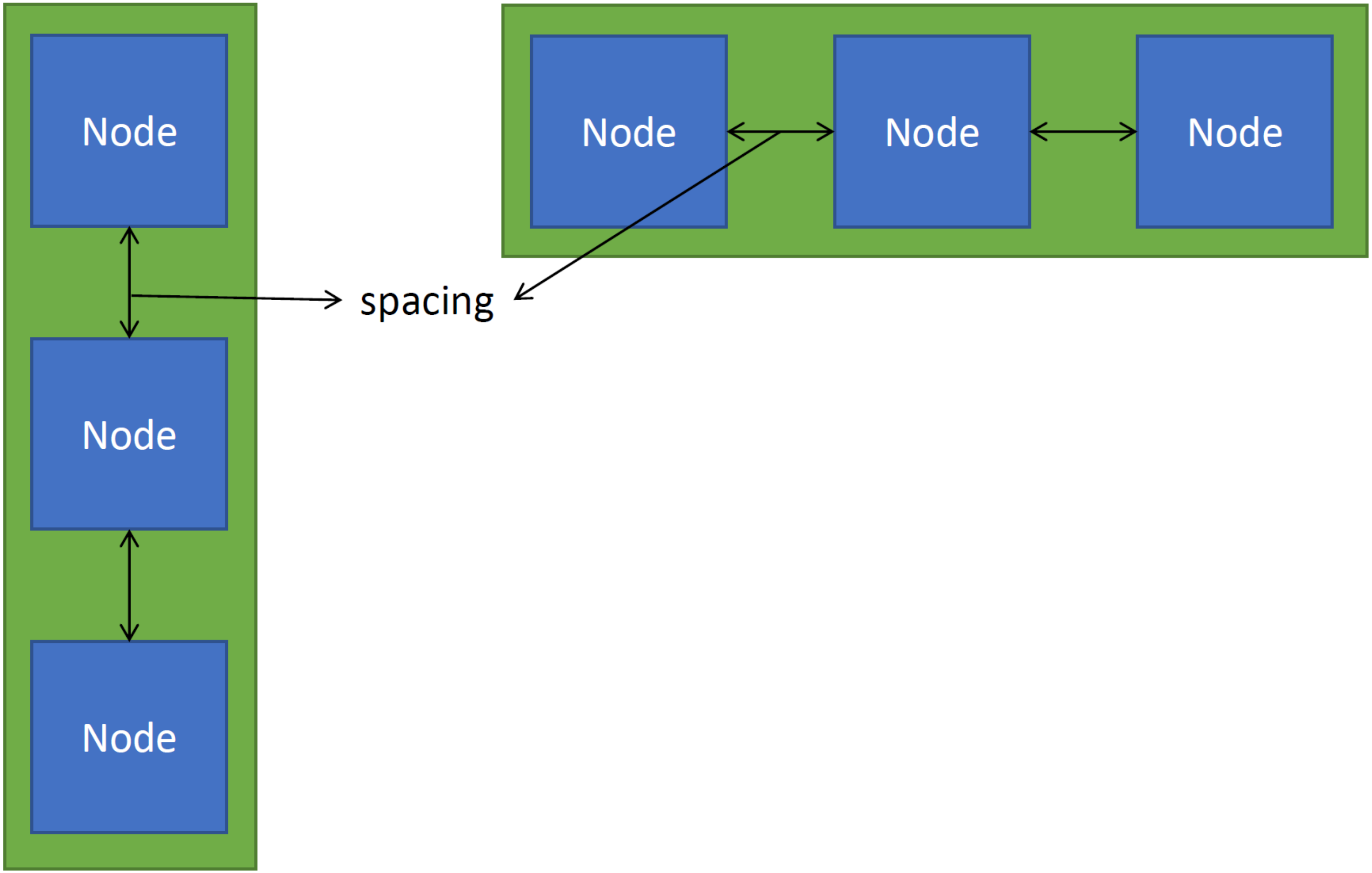
- Layout classes are called **Panes** in JavaFX
- Layout Pane automatically **controls** the **size** and **placement** of components in a container
  - Frees programmer from handling/hardcoding positioning of UI elements
  - As the window is resized, the UI components reorganize themselves based on the rules of the layout

Common Layouts

- **VBox** - displays UI elements in a vertical line
- **HBox** - displays UI elements in a horizontal line
- **BorderPane** - provides five areas: top, left, right, bottom, and center.
- **FlowPane** - lays out its child components either vertically or horizontally. Can wrap the components onto the next row or column if there is not enough space in a row/column.
- **GridPane** - displays UI elements in a grid (e.g., a grid of 2 rows by 2 columns)

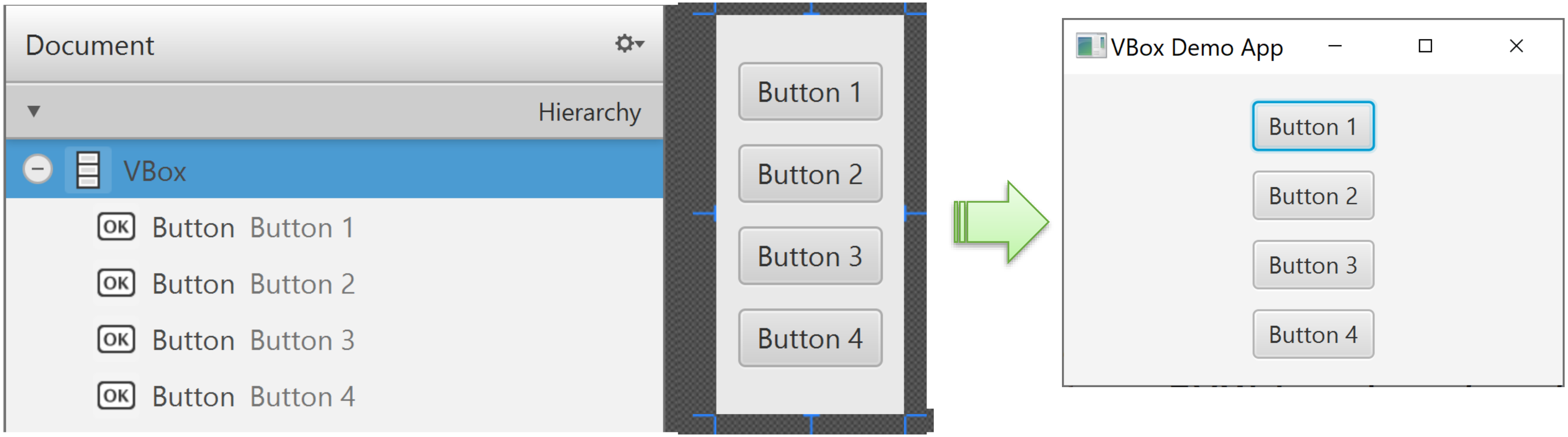


VBox & HBox



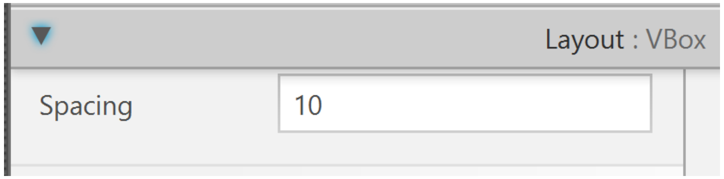
VBox Example

- **VBox** pane creates an easy layout for arranging child components in a *single vertical column*
  - Create a VBox layout container
  - Add 4 buttons to the VBox



Customizing VBox layout

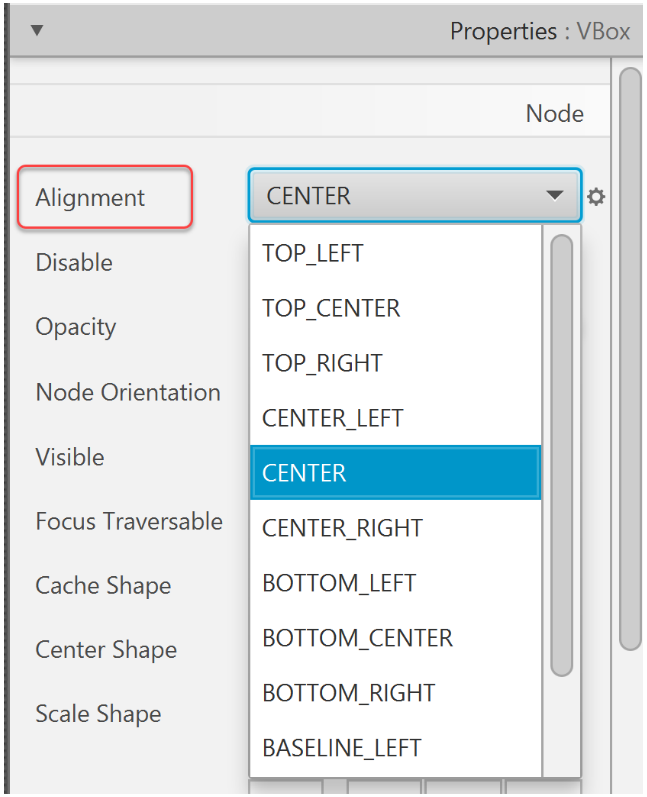
- We can customize vertical spacing *between* children using VBox's **Spacing** property



Layout : VBox

Spacing 10

- Can also alignment of child components
  - Default positioning is in TOP_LEFT (Top Vertically, Left Horizontally)
  - Can change Vertical/Horizontal alignment
    - e.g. BOTTOM_RIGHT represents alignment on the bottom vertically, right horizontally

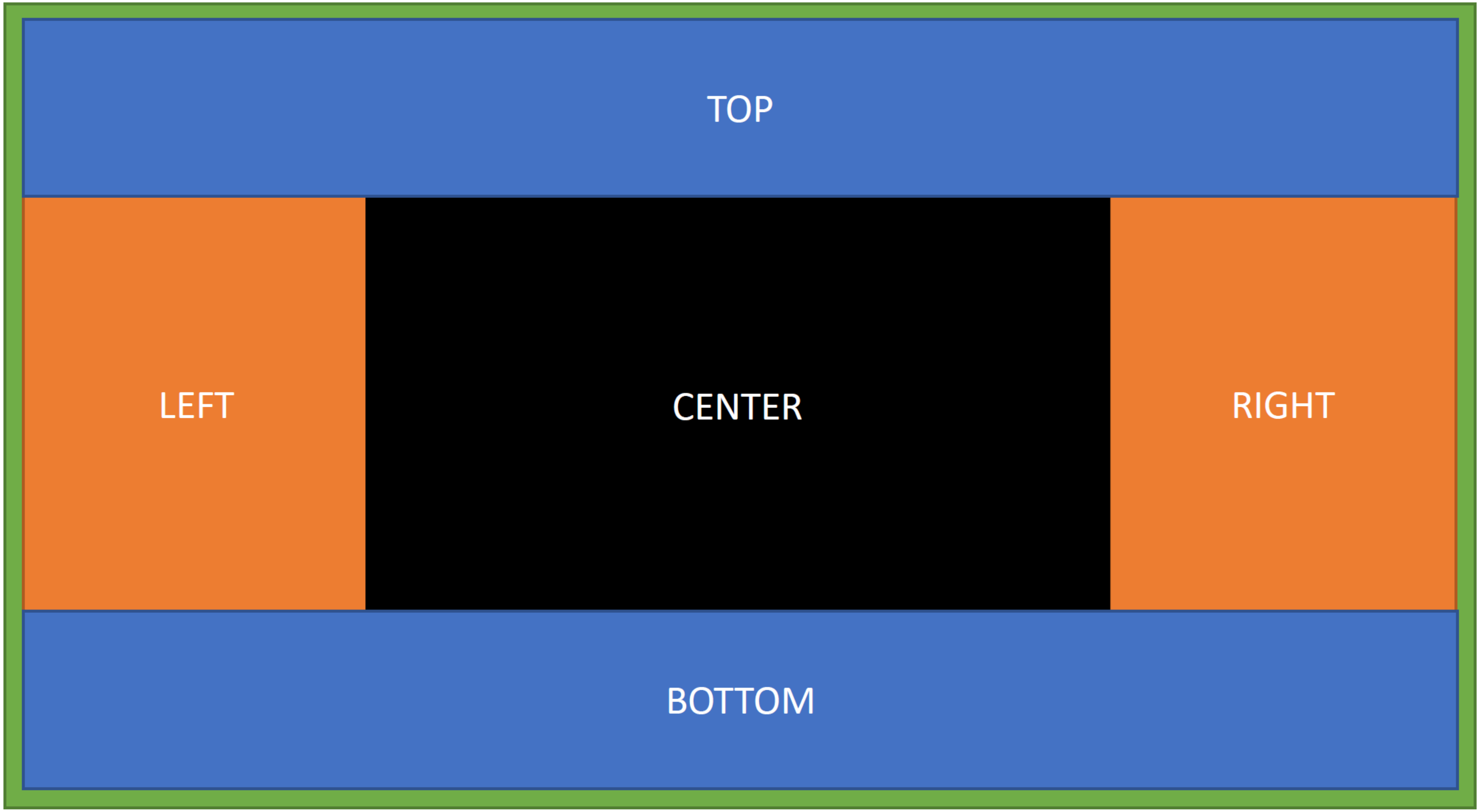


Properties : VBox

Node

Alignment CENTER

BorderPane



TOP

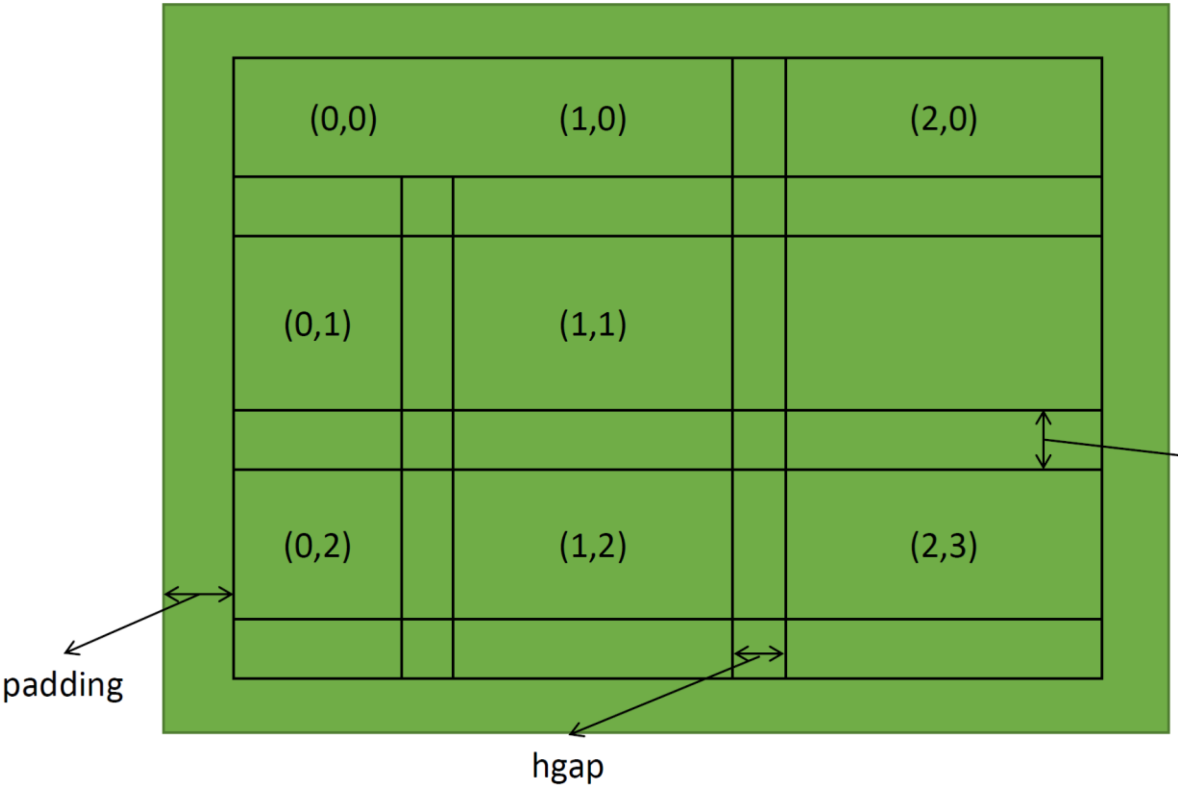
LEFT

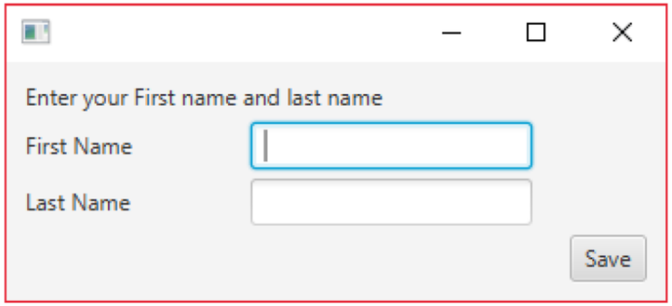
CENTER

RIGHT

BOTTOM

GridPane





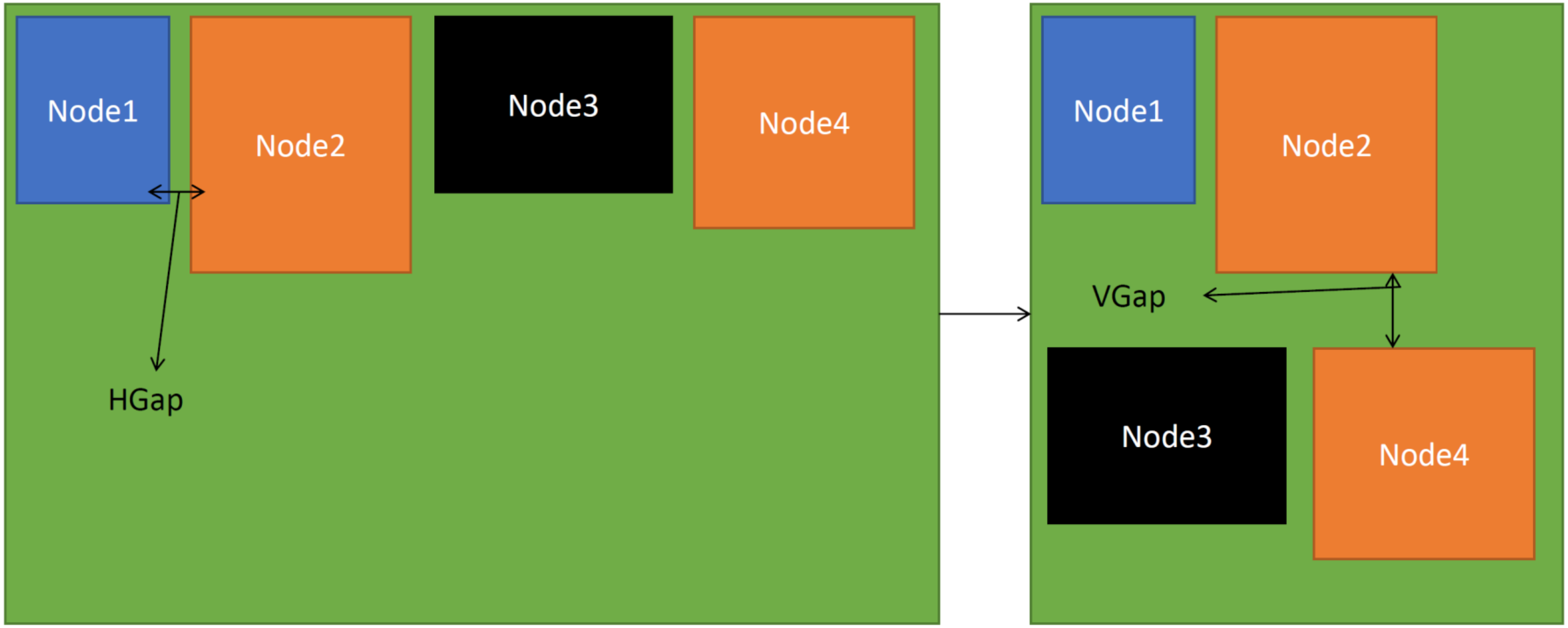
Enter your First name and last name

First Name

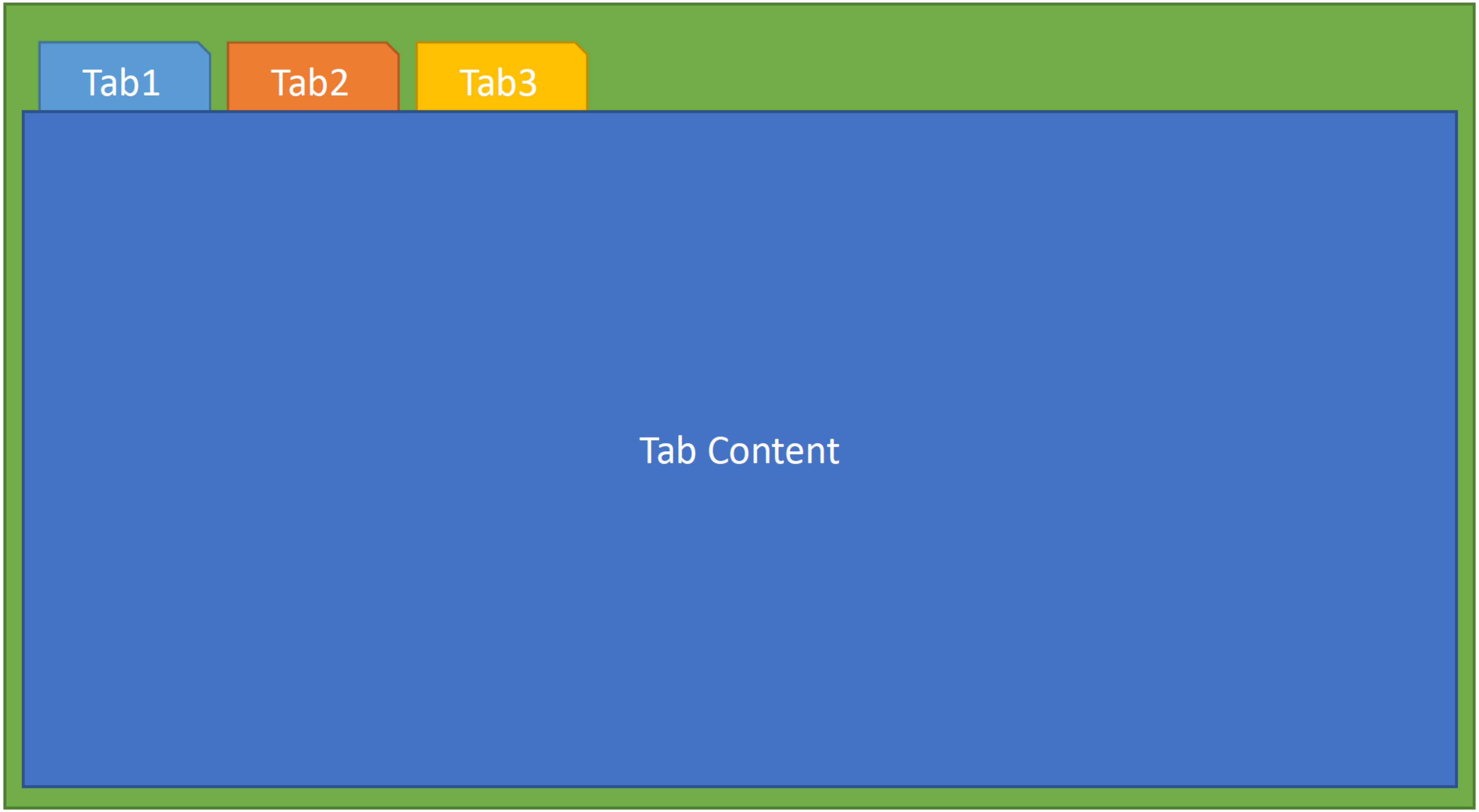
Last Name

FlowPane

- With **FlowPane** the components are arranged from left to right and top to bottom manner in the order they were added

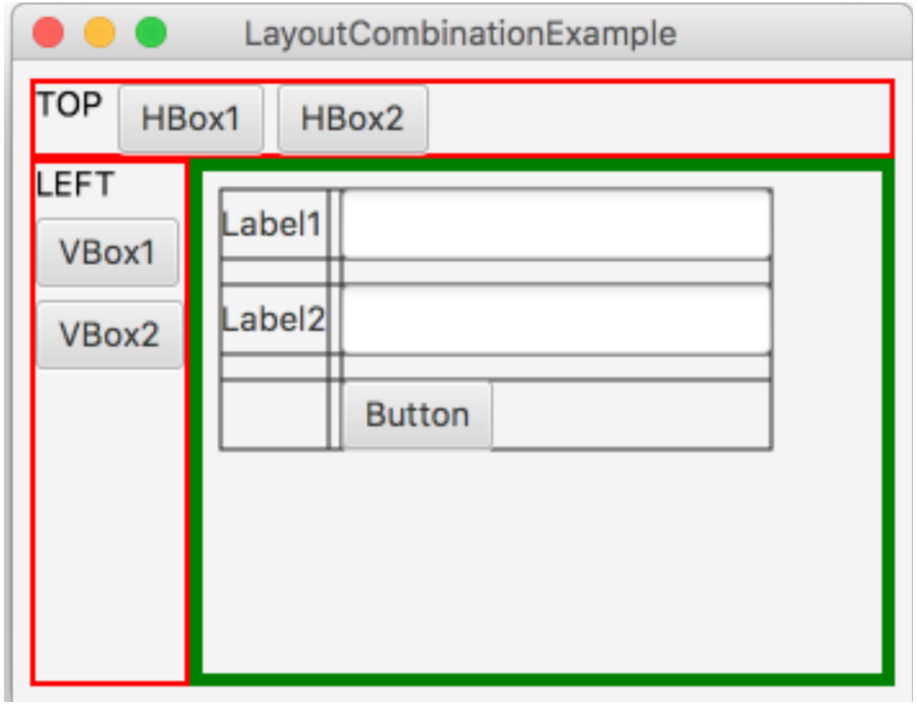


TabPane



Complex Layouts

- For more complex views you can combine different layouts to group components
  - e.g., a BorderPane that contains VBox and HBox panes



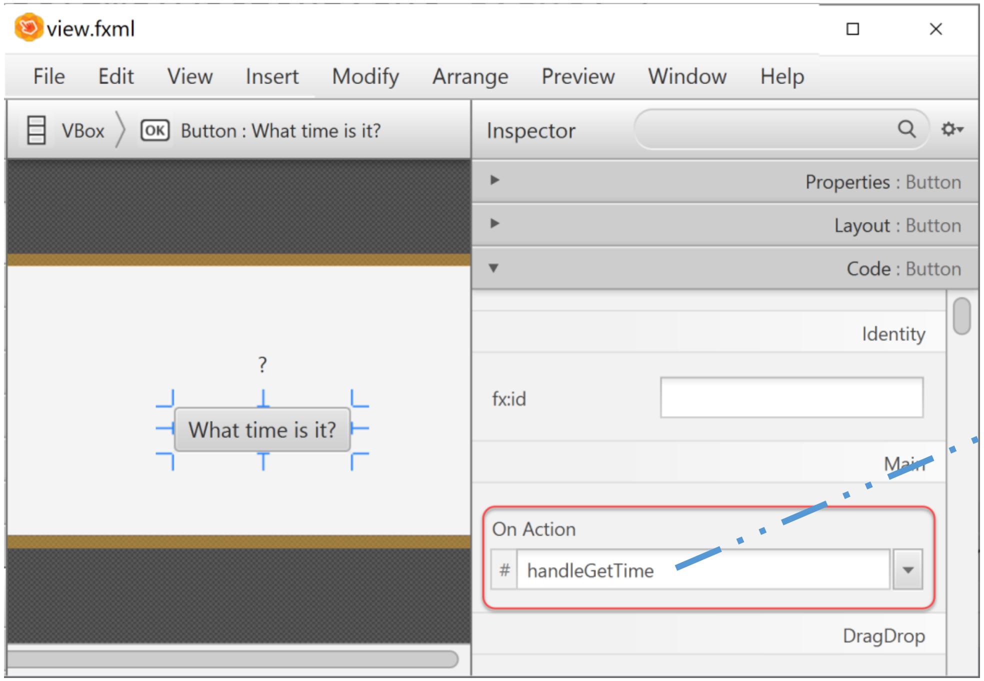
Handling Events

< Back

What is Event Driven Programming?

- GUI programming model is based on **event driven programming**
- Code is executed upon activation of events
- An **event** is a signal that some something of interest to the application has occurred
  - Keyboard (key press, key release)
  - Mouse Events (clicked, mouse enters, mouse leaves)
  - Input focus (gained, lost)
  - Window events (starting, closing, maximize, minimize)
- When an event is triggered, an event handler can run to respond to the event. e.g.,
  - When the button is clicked -> load the data from a file into a list

Set the Event Handler name in the view using Scene Builder them implement it in the Controller



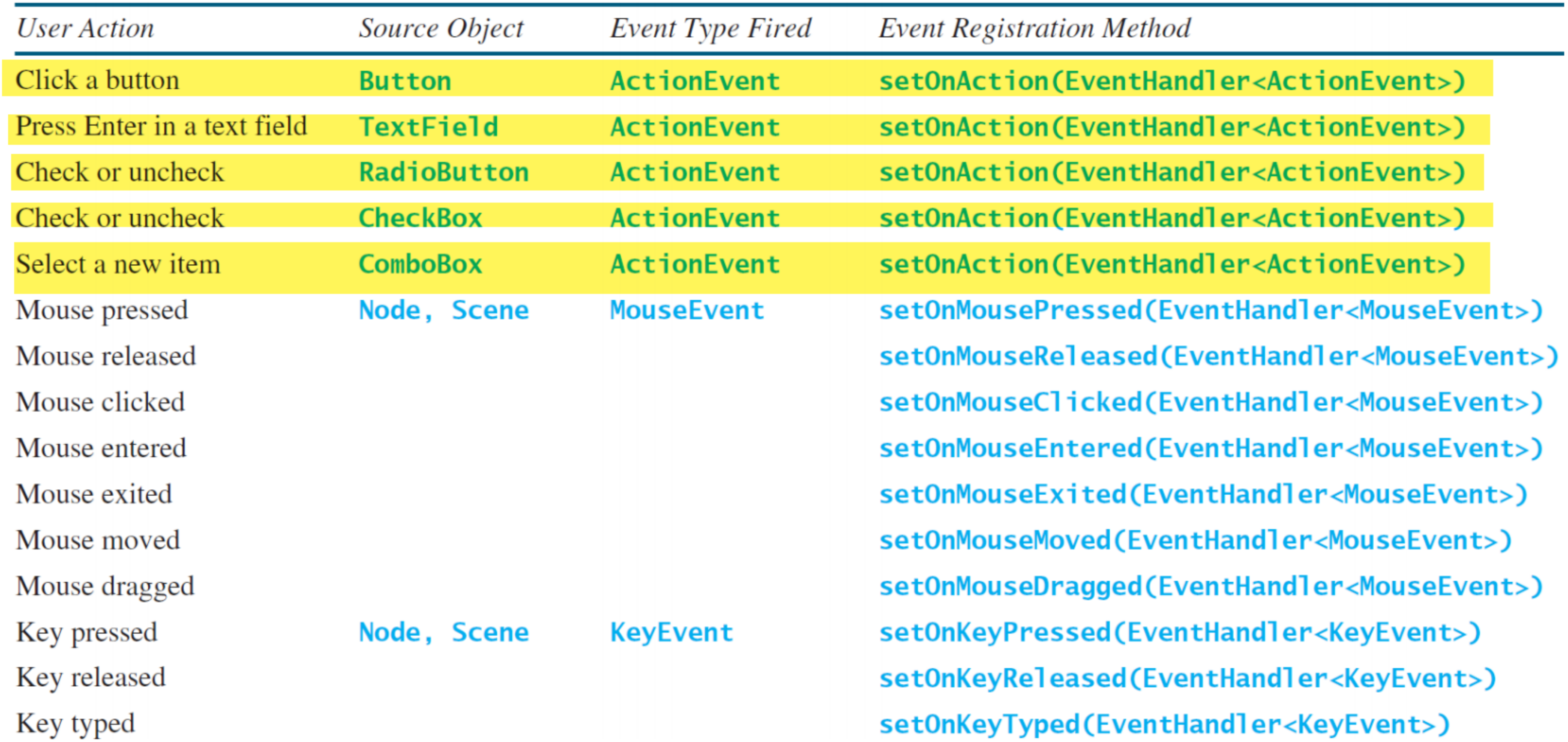
fx:id

On Action: handleGetTime

```
public class Controller {
@FXML private Label timeLabel;

 @FXML
 void handleGetTime (ActionEvent event) {
     timeLabel.setText(
        Model.getCurrentDateTime());
 }
}
```

- **ActionEvent** is the most commonly used event to handle button clicks and selection changes of dropdowns and lists.
- The event object contain information about the event such as the event source (e.g., button that was clicked) and the event type (e.g., click event).

User Actions and Corresponding Event



User Action	*Source Object*	*Event Type Fired*	*Event Registration Method*
Click a button	`Button`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Press Enter in a text field	`TextField`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Check or uncheck	`RadioButton`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Check or uncheck	`CheckBox`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Select a new item	`ComboBox`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Mouse pressed	`Node, Scene`	`MouseEvent`	`setOnMousePressed(EventHandler<MouseEvent>)`
Mouse released			`setOnMouseReleased(EventHandler<MouseEvent>)`
Mouse clicked			`setOnMouseClicked(EventHandler<MouseEvent>)`
Mouse entered			`setOnMouseEntered(EventHandler<MouseEvent>)`
Mouse exited			`setOnMouseExited(EventHandler<MouseEvent>)`
Mouse moved			`setOnMouseMoved(EventHandler<MouseEvent>)`
Mouse dragged			`setOnMouseDragged(EventHandler<MouseEvent>)`
Key pressed	`Node, Scene`	`KeyEvent`	`setOnKeyPressed(EventHandler<KeyEvent>)`
Key released			`setOnKeyReleased(EventHandler<KeyEvent>)`
Key typed			`setOnKeyTyped(EventHandler<KeyEvent>)`

The first 5 are the most common events and can be handled as **`ActionEvent`**

Handling Events Programmatically using Lambdas

```
btn.setOnAction(event ->
        handleEvent(event) );

// Or use method reference
btn.setOnAction(this::handleEvent);

private void handleEvent(ActionEvent event) {
    System.out.println(event);
}
```