CMPS 251

Read Chapter 12

Graphical User Interfaces (GUI)

Dr. Abdelkarim Erradi

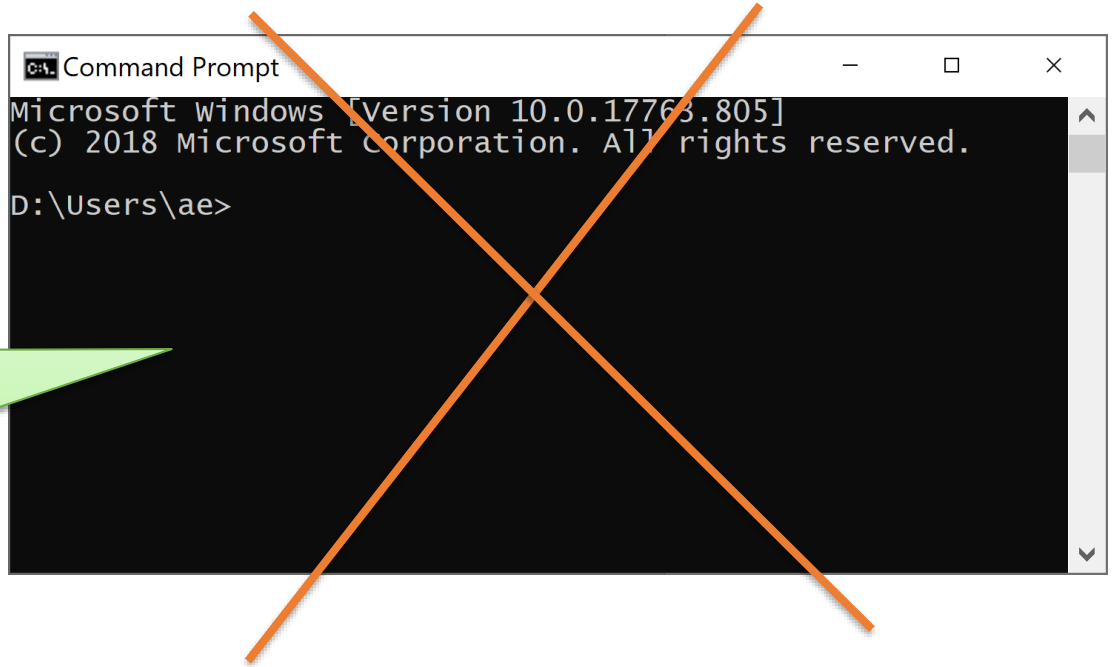
CSE@QU

Outline

1. GUI Programming Model
2. Model-View-Controller (MVC) Pattern
3. Handling Events
4. JavaFX Layouts

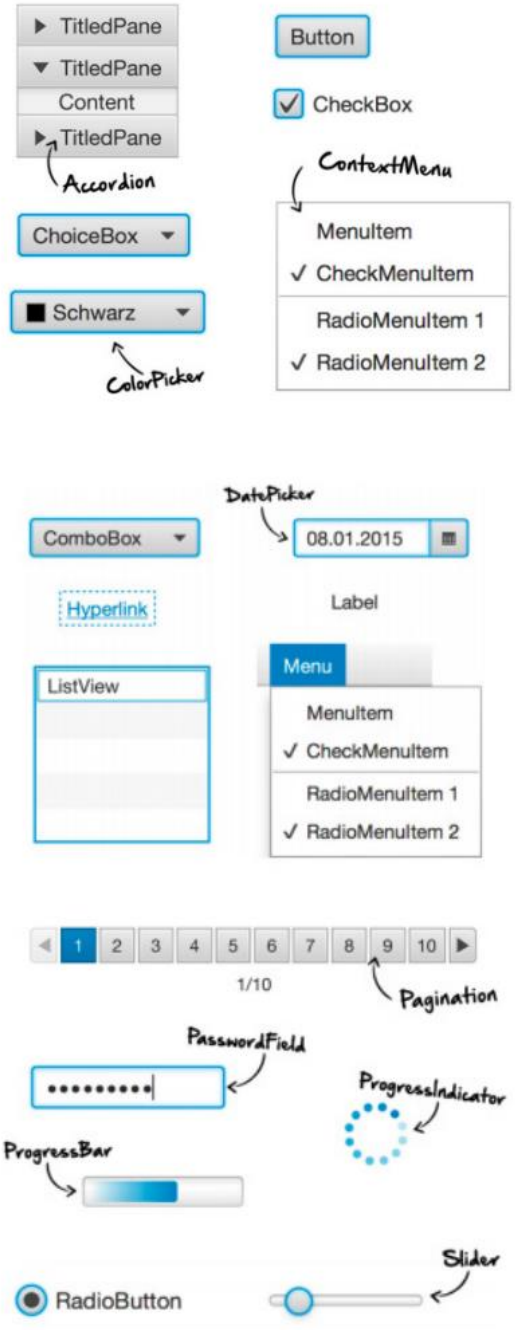
GUI Programming Model

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



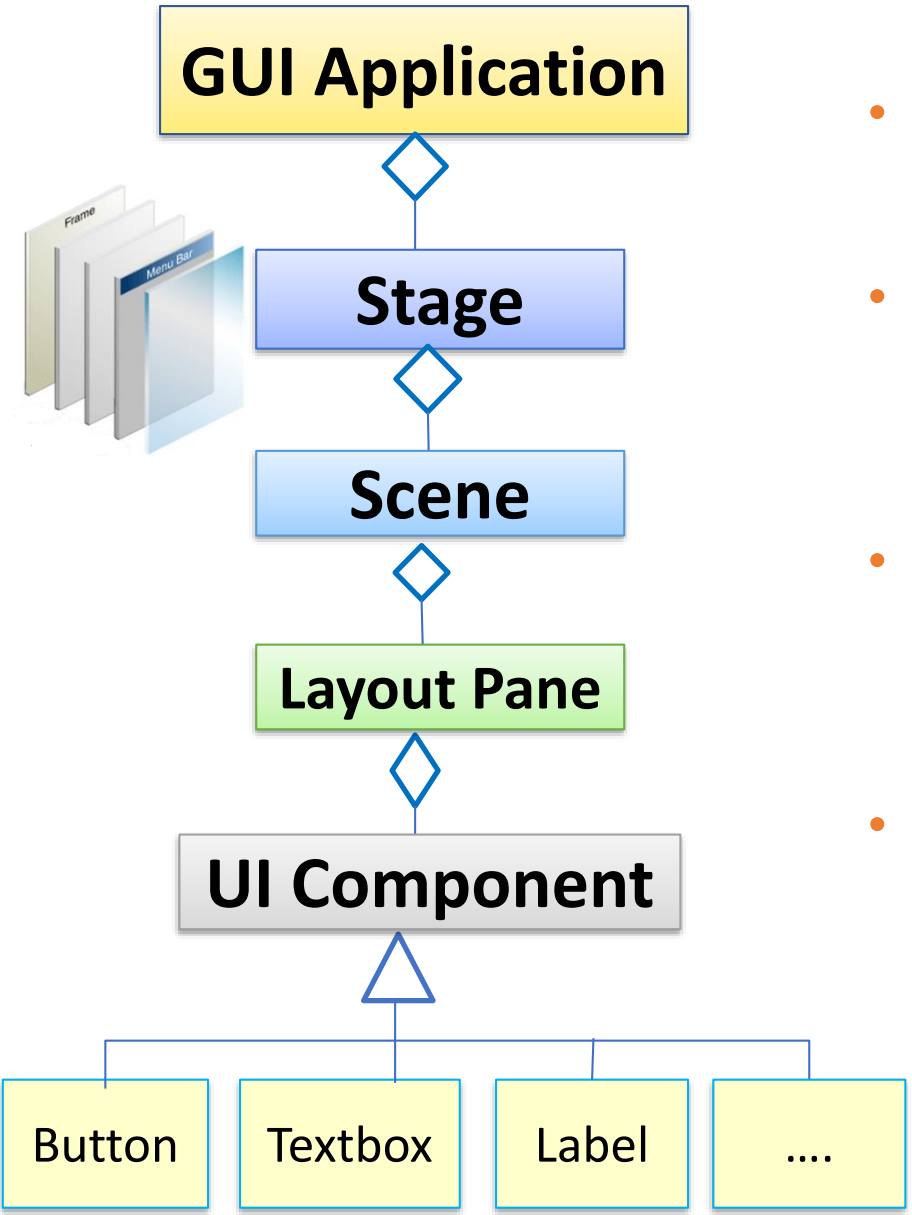
Back

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

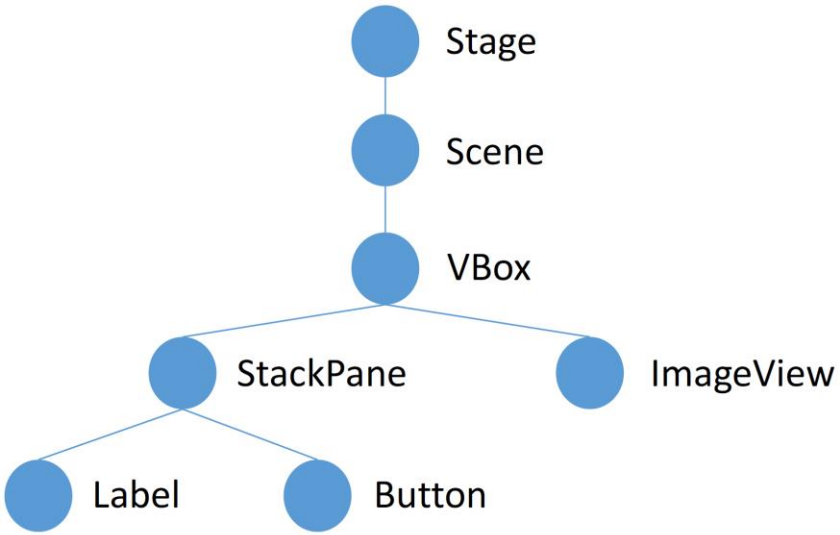


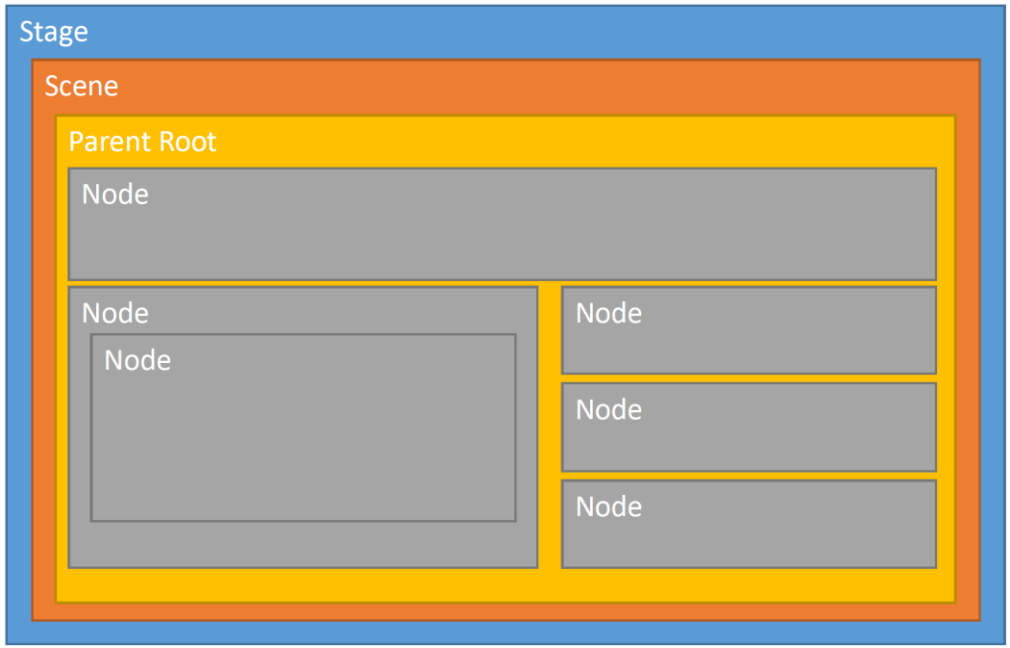
- GUI of an application is made up of **Windows** (called **Stage** in JavaFX)
- A window has a **container** (called **Scene**) to host the UI root layout container
- UI Components are first added to a root layout container (such as VBox) then placed in the Scene
- UI Components **raise Events** when 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

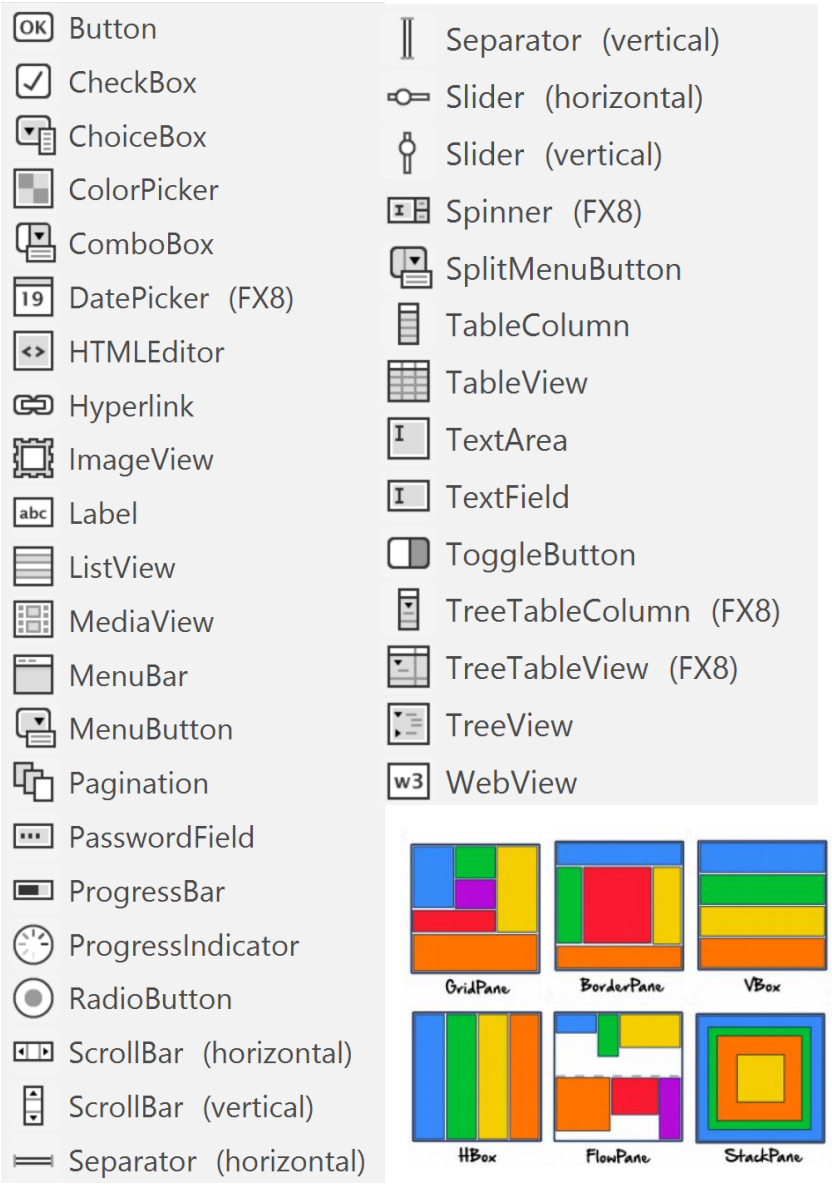
Stage = **Window** where a scene is displayed

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



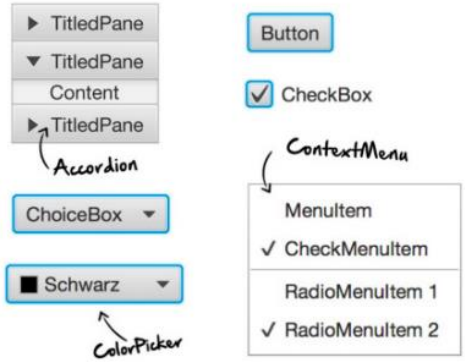


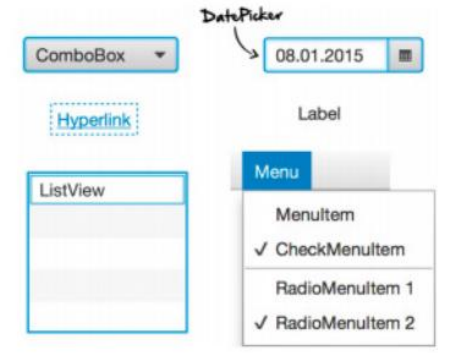
What Makes up JavaFX ?

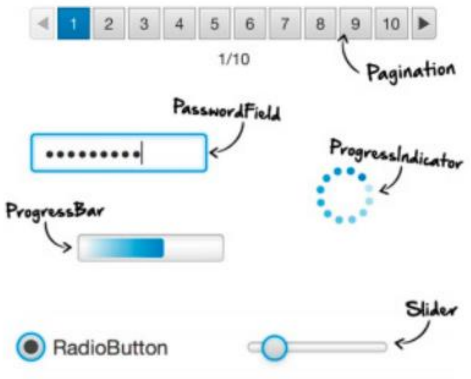


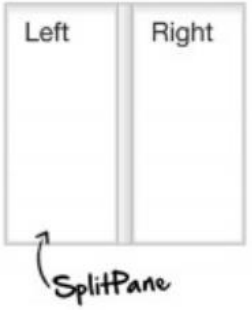
- **UI components**
  - Set of pre-built UI components that can be composed to create a GUI
  - e.g. buttons, text-fields, menus, tables, lists, etc.
- **Layout containers**
  - Control placement/ positioning of components in the form (e.g., VBox and HBox)

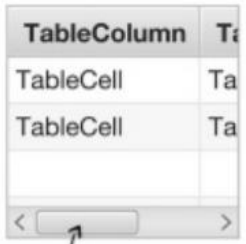
JavaFX UI Components

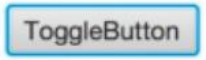


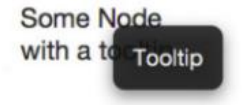


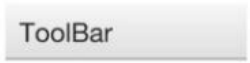


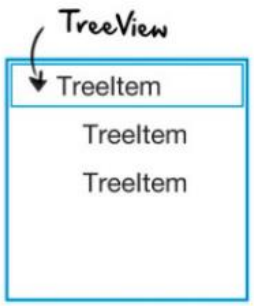


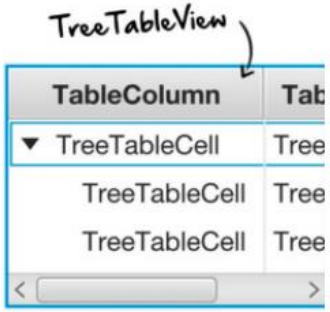




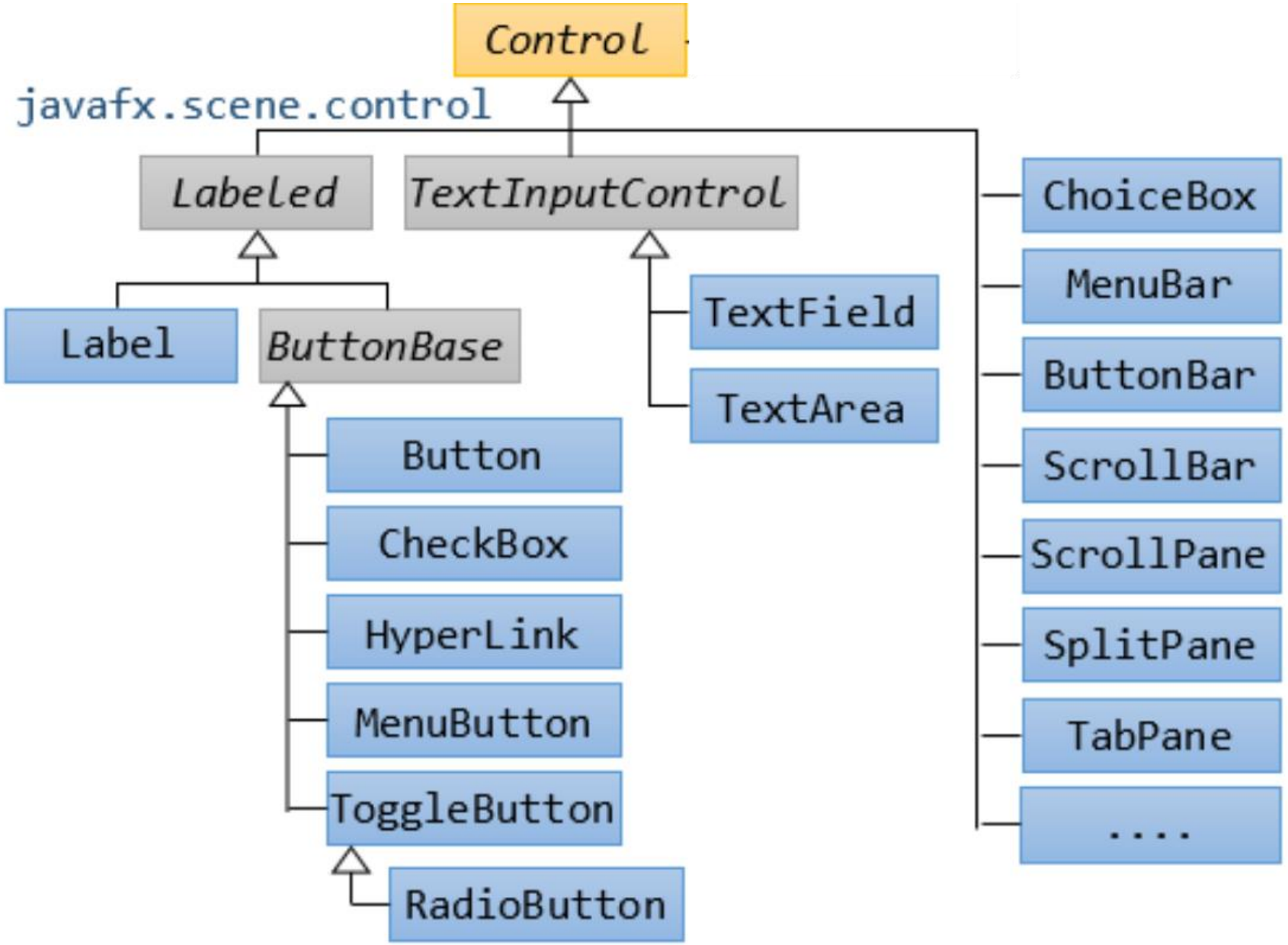








JavaFX UI Components Hierarchy

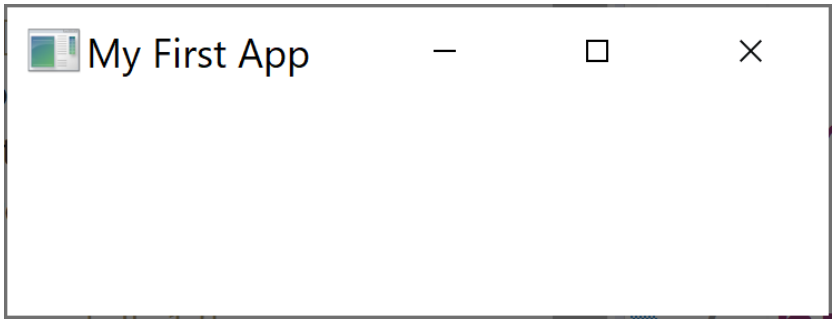


Creating JavaFX GUI: Stage (1/2)

- 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

```
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);
 }
}
```



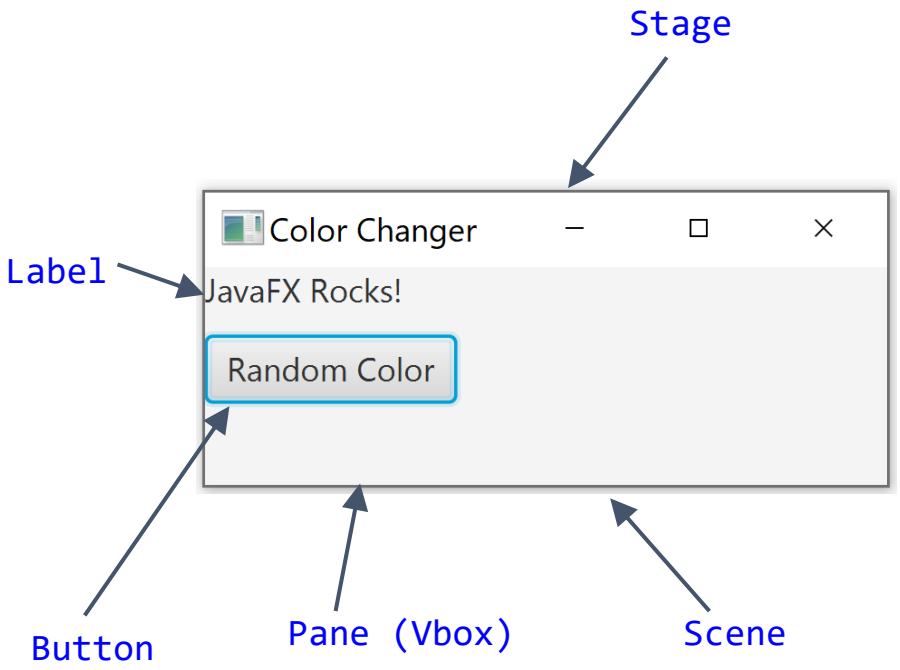
Creating JavaFX GUI : Scene (2/2)

- 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

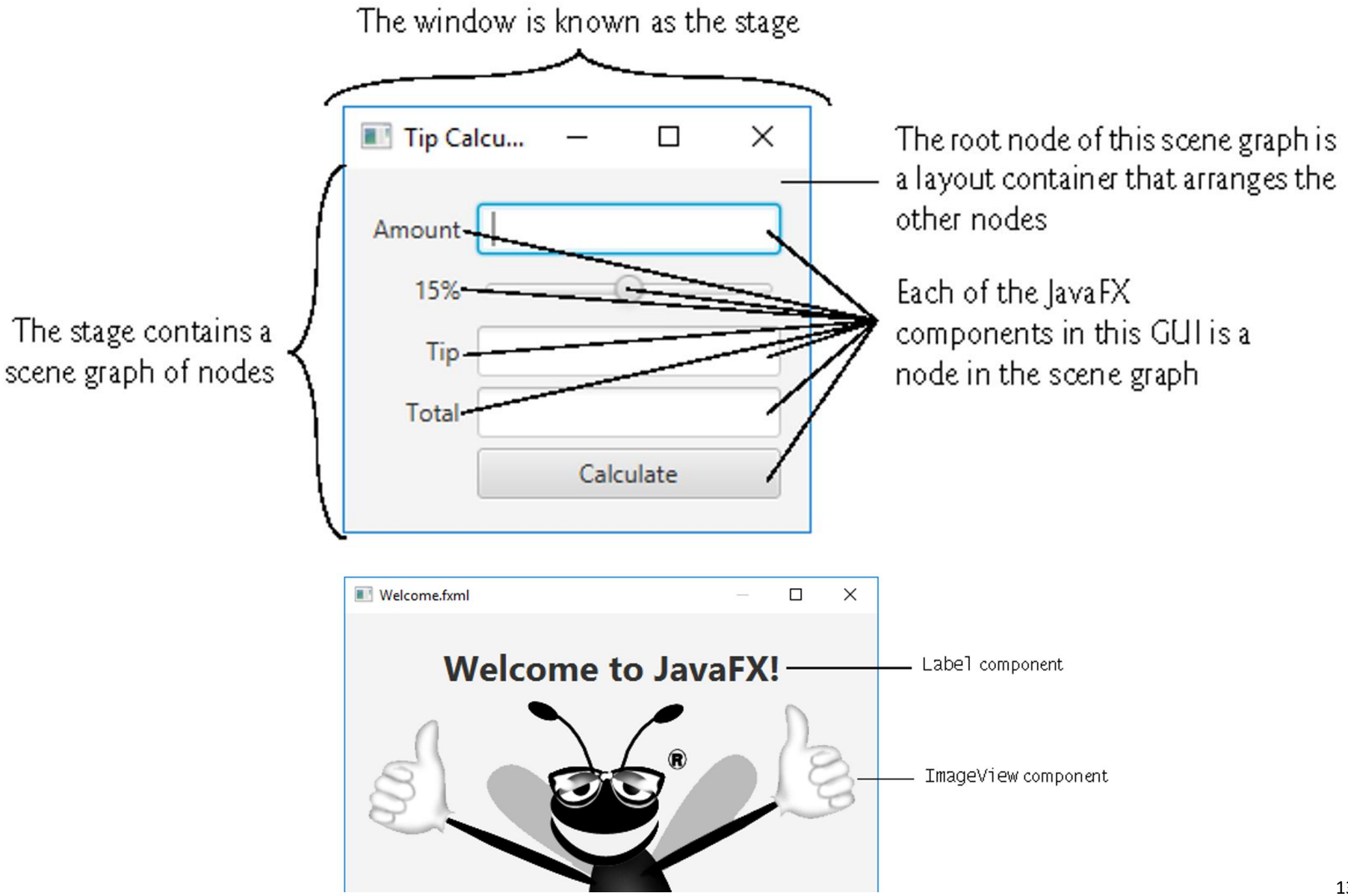
```
public void start(Stage stage) {
   VBox root = new VBox();
   Label label = new Label("JavaFX Rocks!");
   Button button = new Button("Submit");
   root.getChildren().addAll(label, button);
   Scene scene = new Scene(root, 200, 200);
   stage.setScene(scene);
   stage.show();
}
```

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



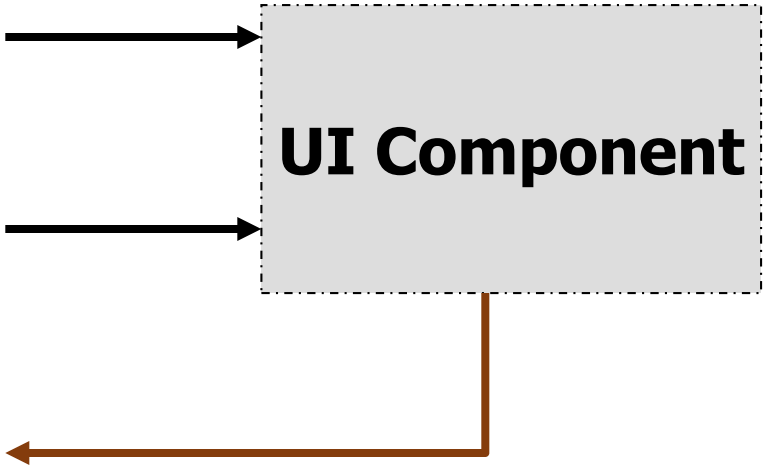
UI Component

- UI component is a class that has:

Attributes

Methods

Events



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**

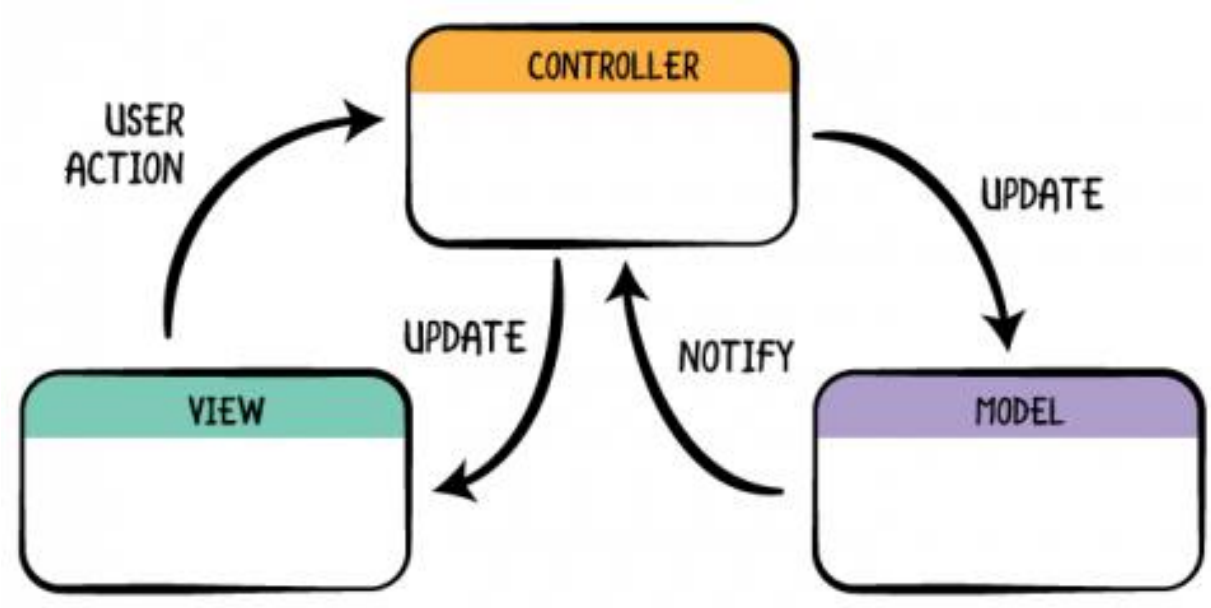
```
vBox.add(button);
```

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) );
```

Model-View-Controller (MVC) Pattern



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

View

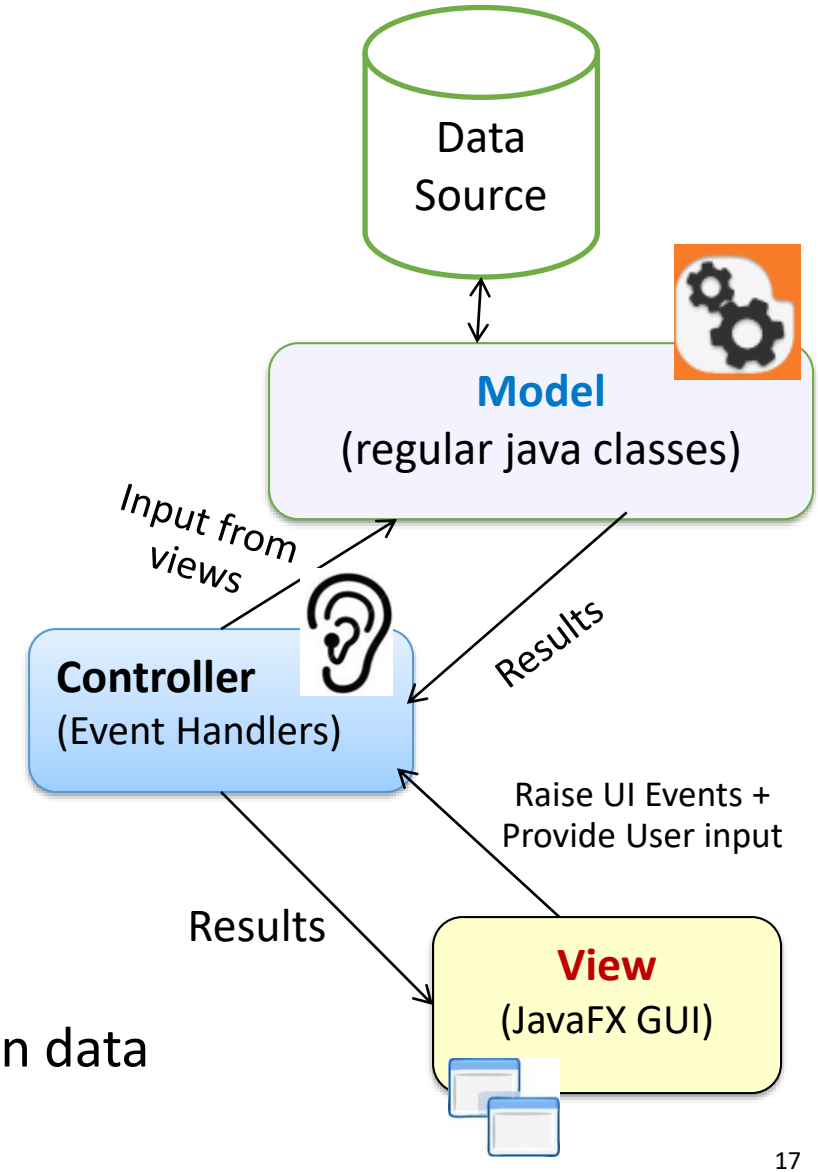
- Gets input from the user
- Notifies the controller about UI events
- Displays output to the user

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

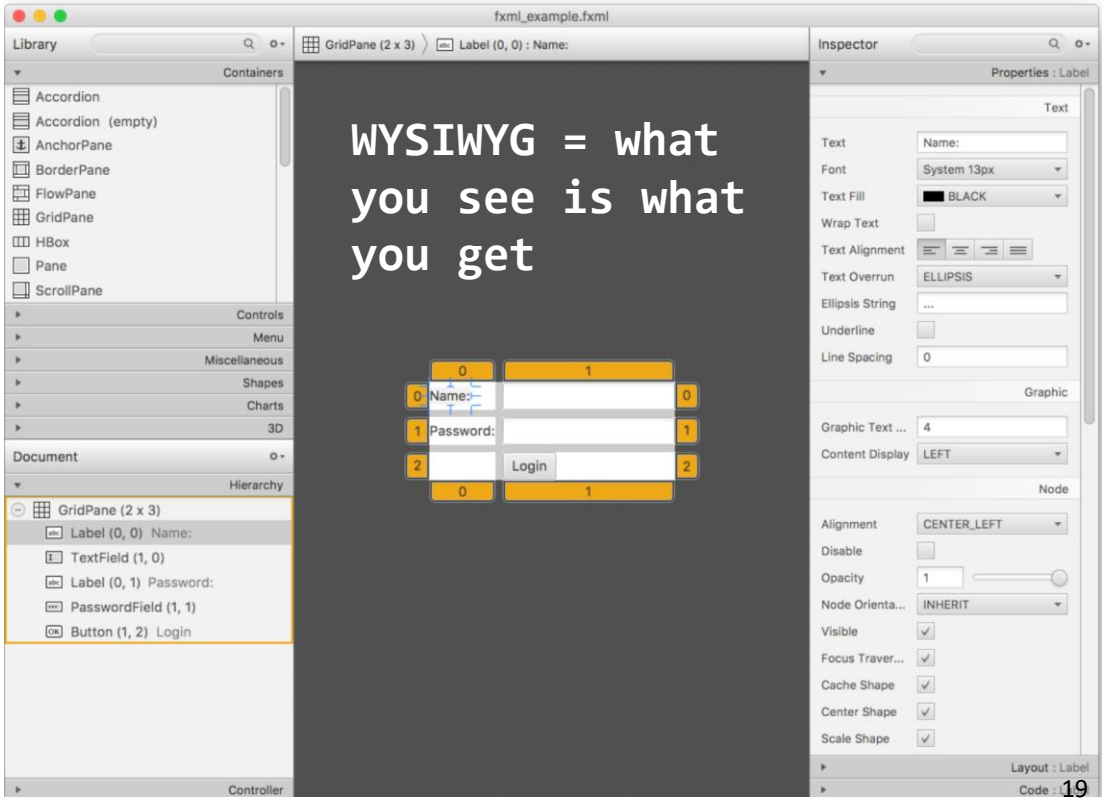
Advantages of MVC

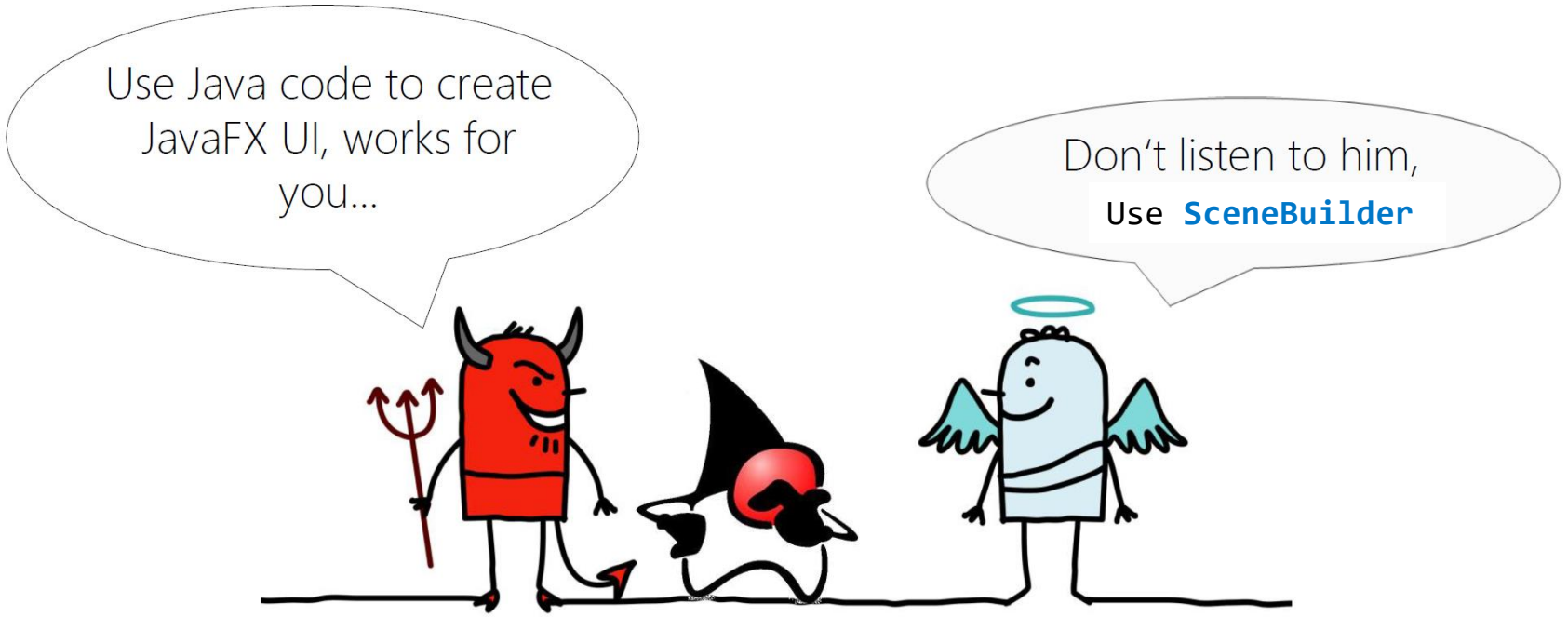
- ***Separation of concerns***
  - Views, controller, and model are **separate components**
    - Computation is not intermixed with Presentation. Consequently, code is cleaner, flexible and easier to understand and change.
    - Allow changing a component without significantly disturbing the others (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 and the app logic
- **SceneBuilder** is a WYSIWYG editor for FXML



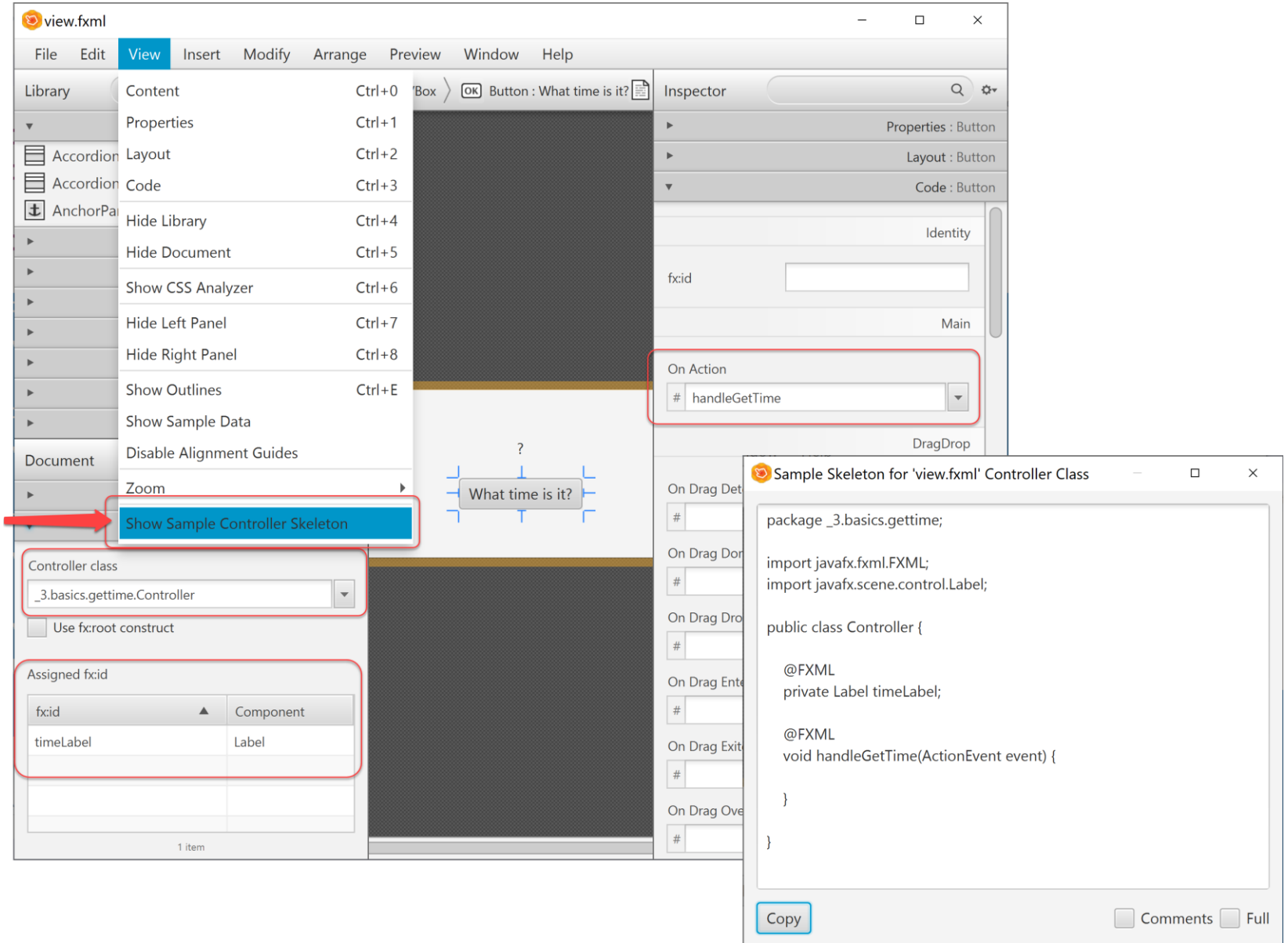



```
VBox root = new VBox();
Label label = new Label("JavaFX Rocks!");
Button button = new Button("Random Color");
button.setTextFill(Color.BLUE);
root.getChildren().addAll(label, button);
root.setSpacing(20);
root.setAlignment(Pos.CENTER);
```

Implementing MVC with JavaFX (1 of 2)

1. Build the **View** using SceneBuilder:
   - Name ONLY the components that will be programmatically accessed (assign the name to the **fx:id** property)
   - Assign **event handler** methods to components raising events that the App cares about (e.g., **On Action** event of a button)
   - Assign the **Controller name** to the View's Controller class property.
   - Generate the Controller Skeleton

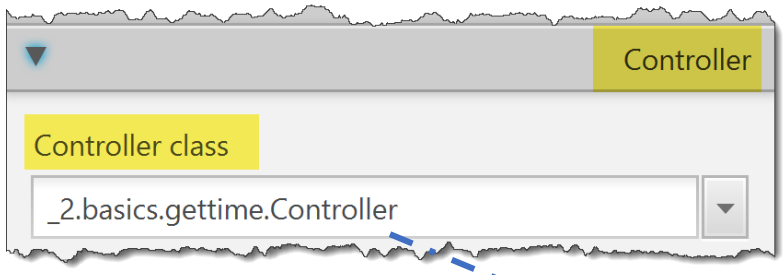
Once you set the fx:id of UI elements and Event Handlers in SceneBuilder you can generate a skeleton Controller class

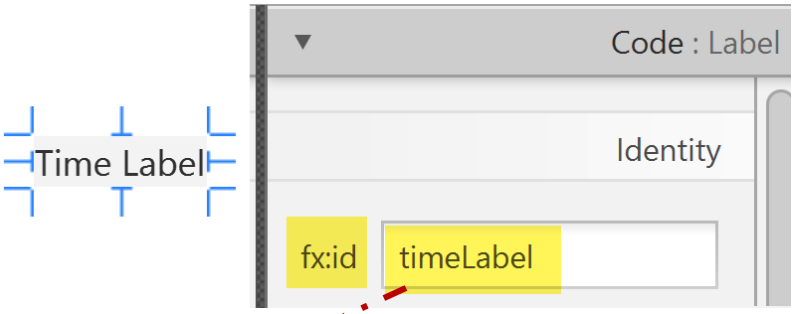


Implementing MVC with JavaFX (2 of 2)

- The View is associated with a **Controller** class that implements the events handlers
- 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 handlers assigned using SceneBuilder
- The controller should call the **Model** to perform computation and get the results

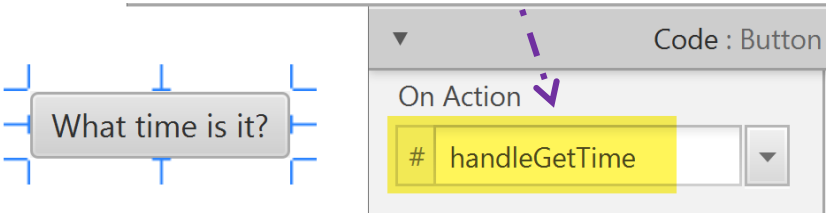
Associating View & Controller






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

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



Launching the App

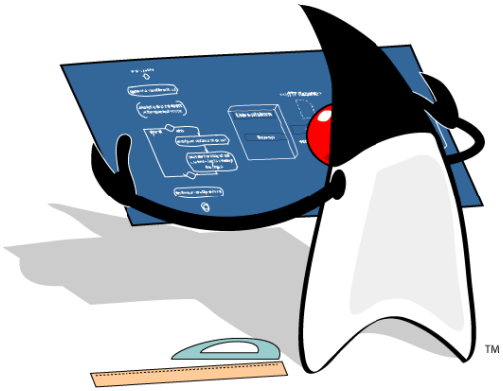
- First load the FXML file of the View in the scene. Then setScene and show the stage.
- This code is the same for any JavaFX app. Just need to change the viewFileName and the windowTitle

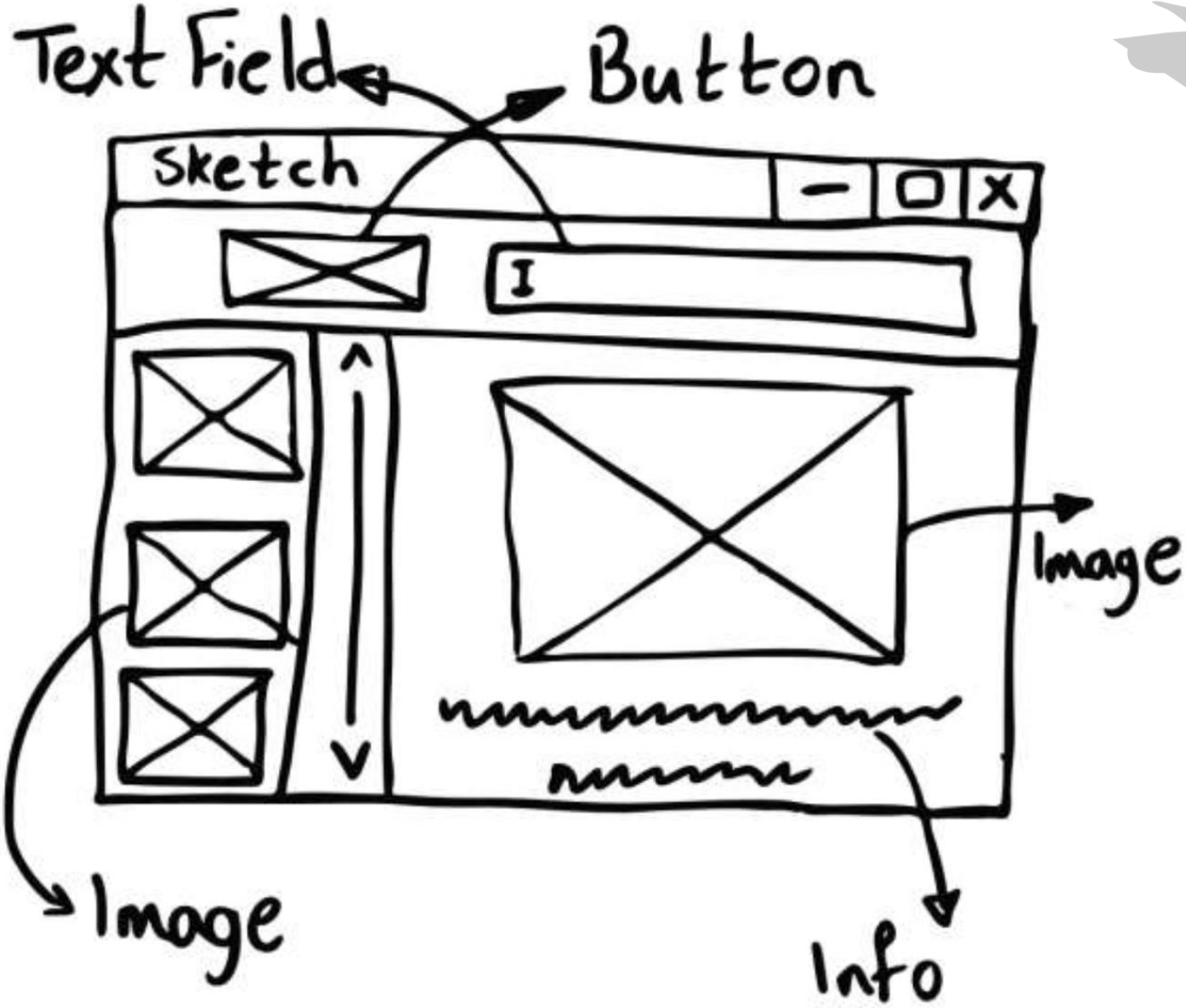
```
 @Override
public void start(Stage stage) throws Exception {
 String viewFileName = "TimeView.fxml";
 String windowTite = "Time App";
 Parent root =
   FXMLLoader.load(getClass().getResource(viewFileName));
 stage.setScene(new Scene(root, 400, 300));
 stage.setTitle(windowTite);
 stage.show();
}
```

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
   - Use layout panes to group and arrange components
3. Add event handlers to respond to the user actions
   - Do something when the user presses a button, selects a combo box element, change text of input field, etc.

UI Sketch - Example



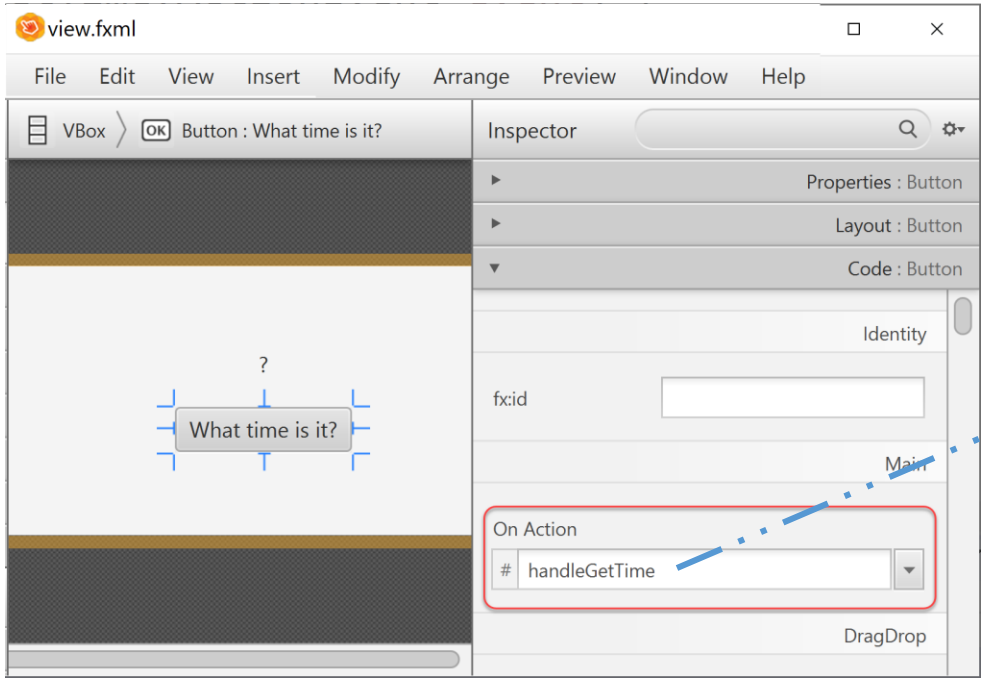


Handling Events

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




```
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`**

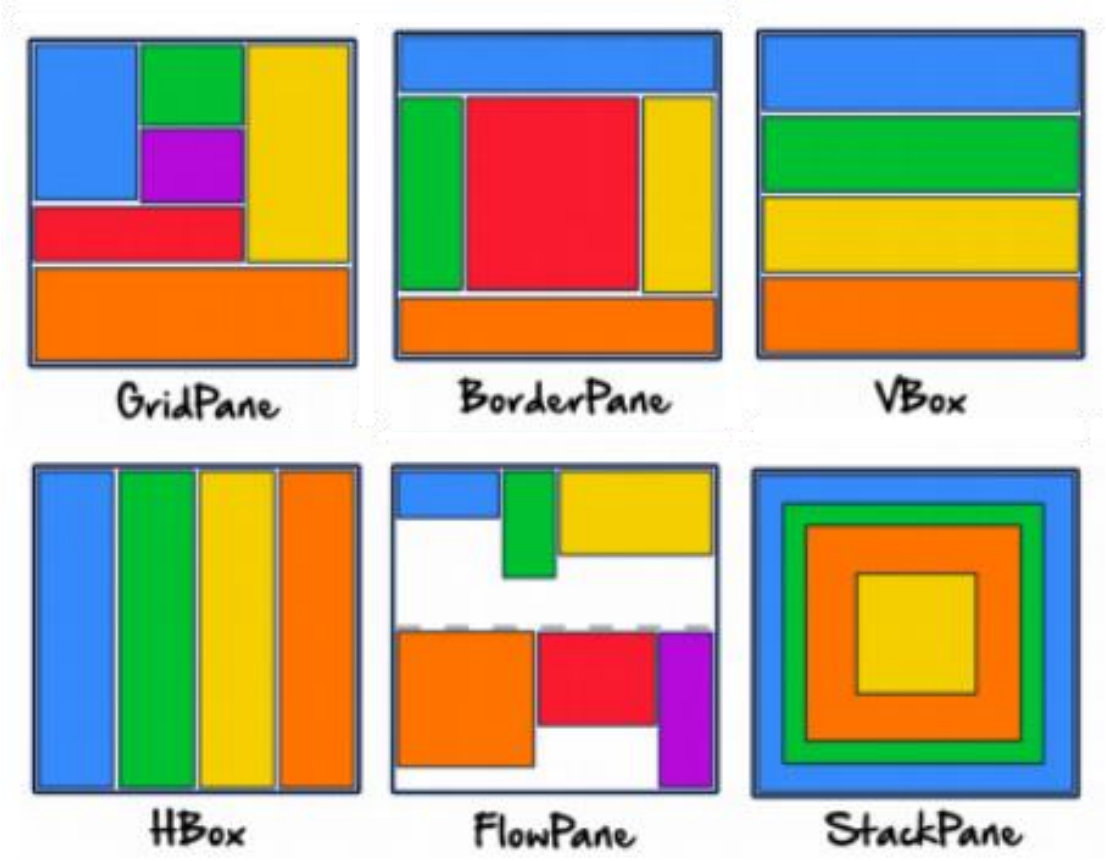
Handling Events Programmatically using Lambdas

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

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

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

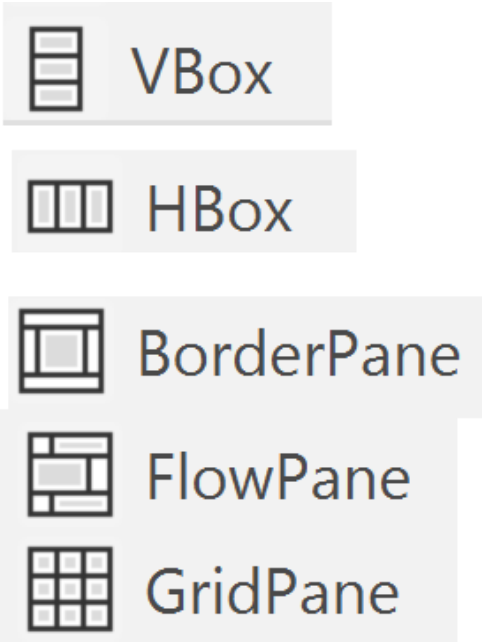
Layouts



Layouts

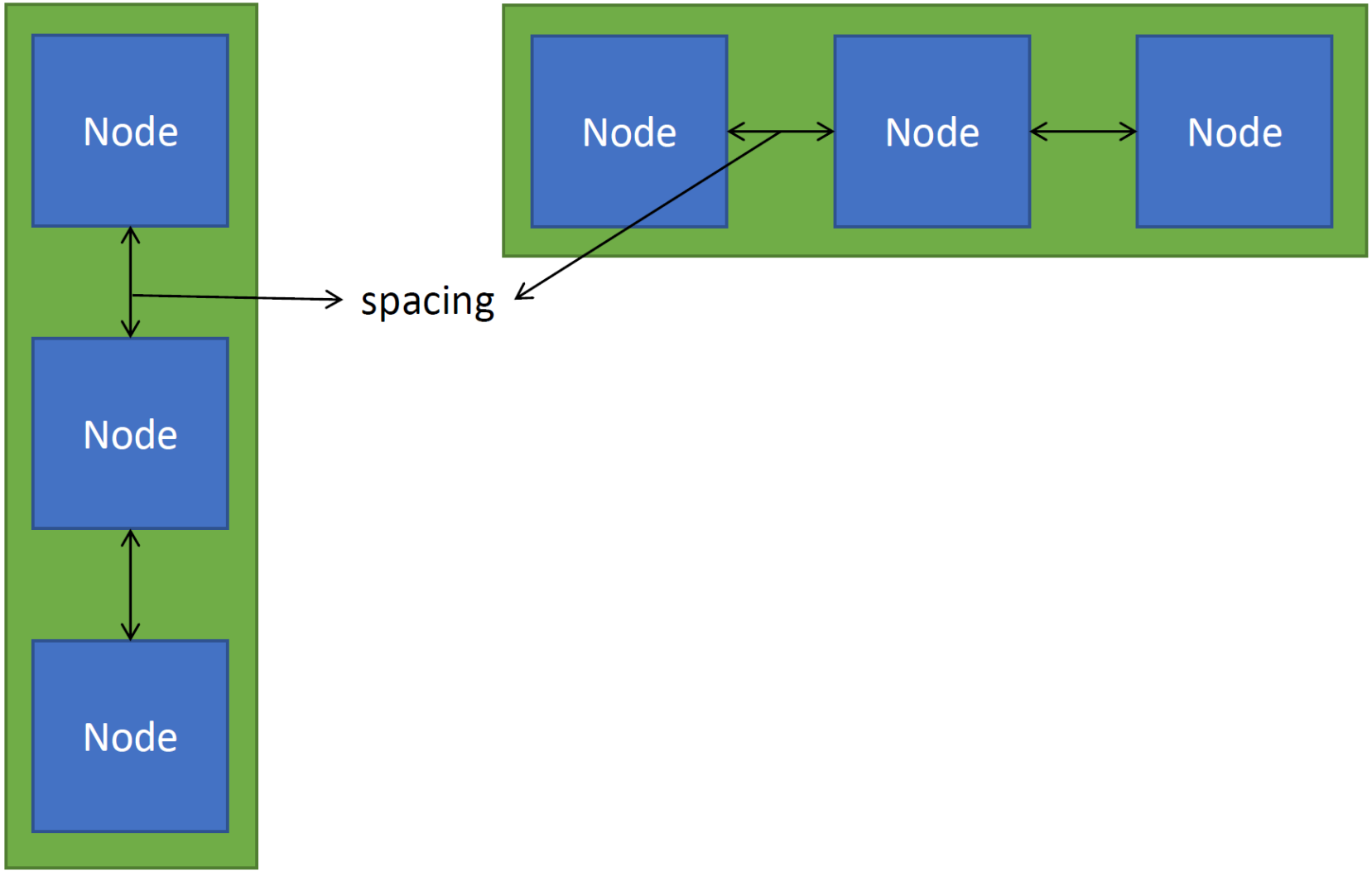
- Layouts are called **Panes** in JavaFX
- Layout Pane automatically **controls** the **size** and **placement** of components in a container to create a **Responsive UI**
  - Frees programmer from handling/hardcoding positioning of UI elements
  - **Responsive UI =** As the window is resized, the UI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

Common Layouts



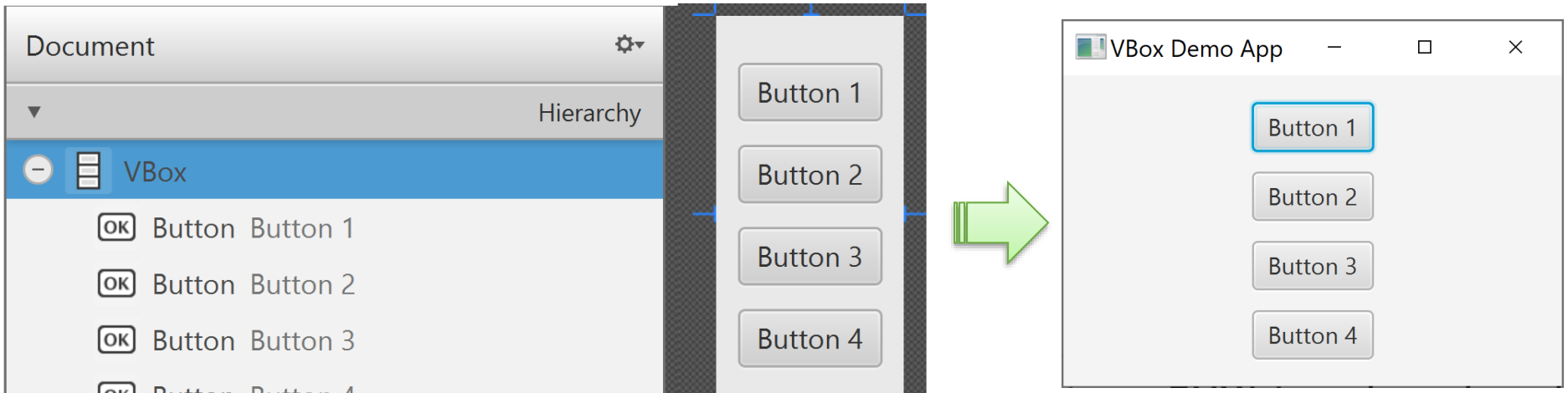
- **VBox** - displays UI elements in a vertical line
- **HBox** - displays UI elements in a horizontal line
- **BorderPane** - provides five areas: top, left, right, bottom, and center.
- **FlowPane** - lays out its child components either vertically or horizontally. Can wrap the components onto the 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



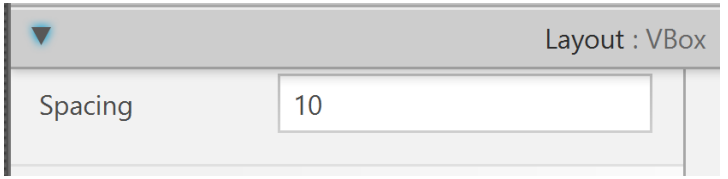
VBox Example

- **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

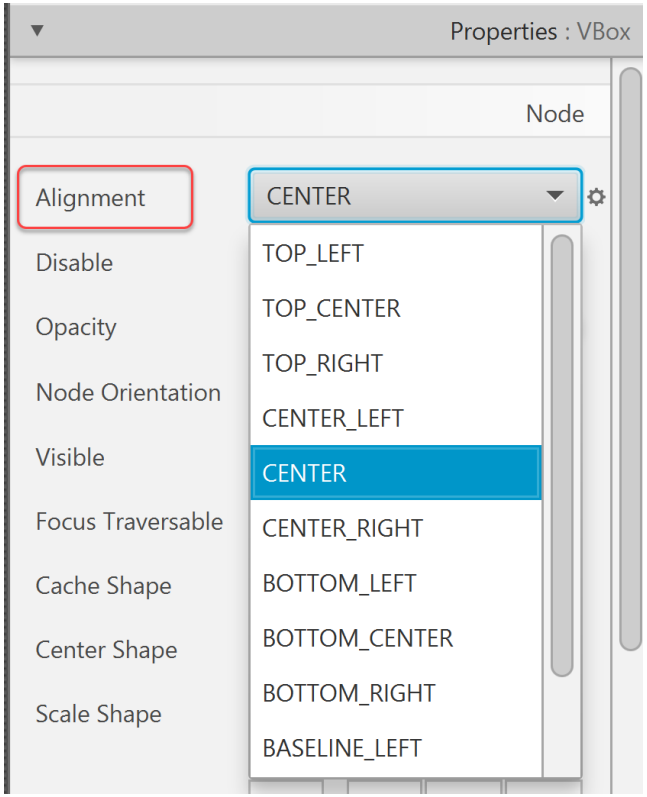


Customizing VBox layout

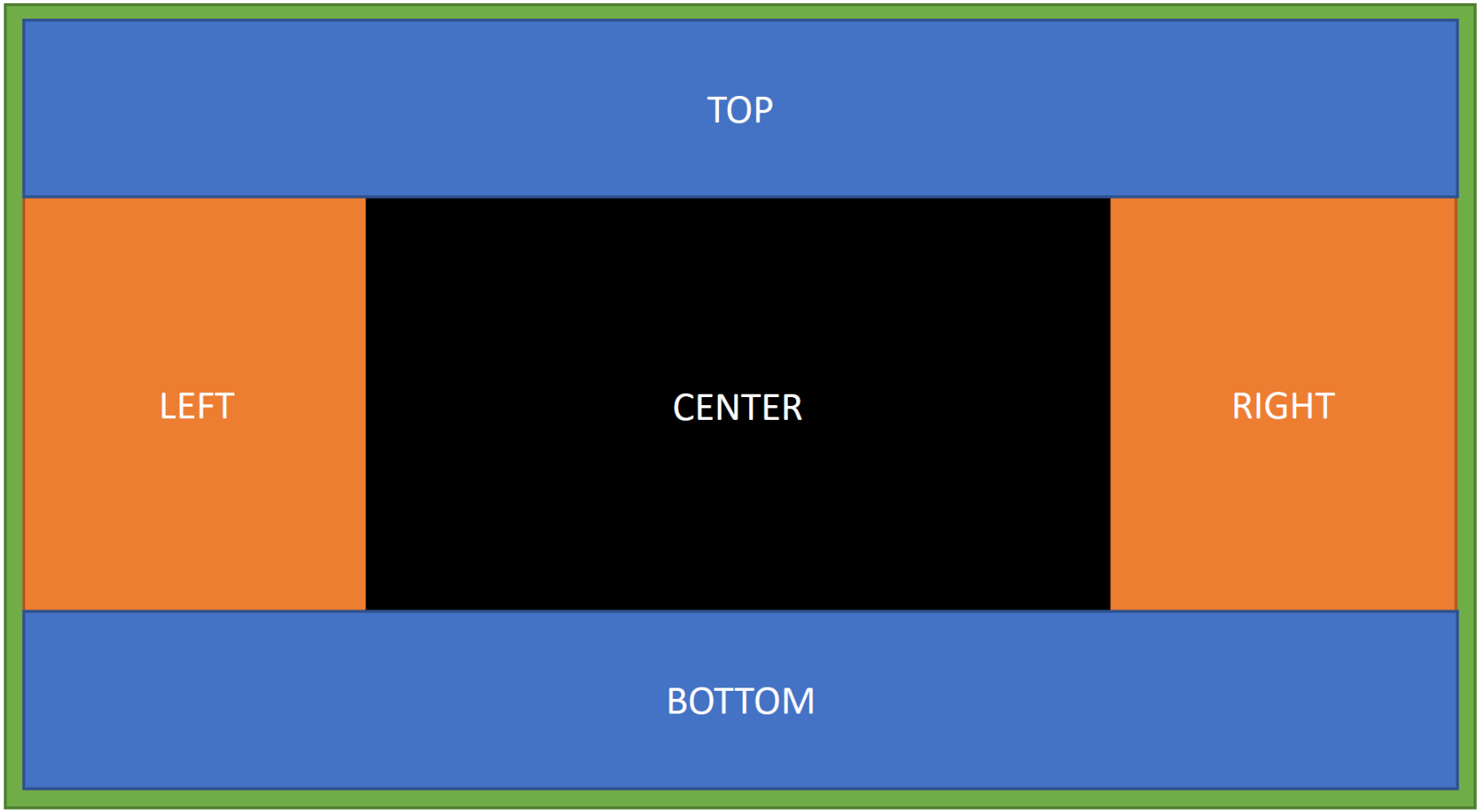
- We can customize vertical spacing *between* children using VBox's `Spacing` property



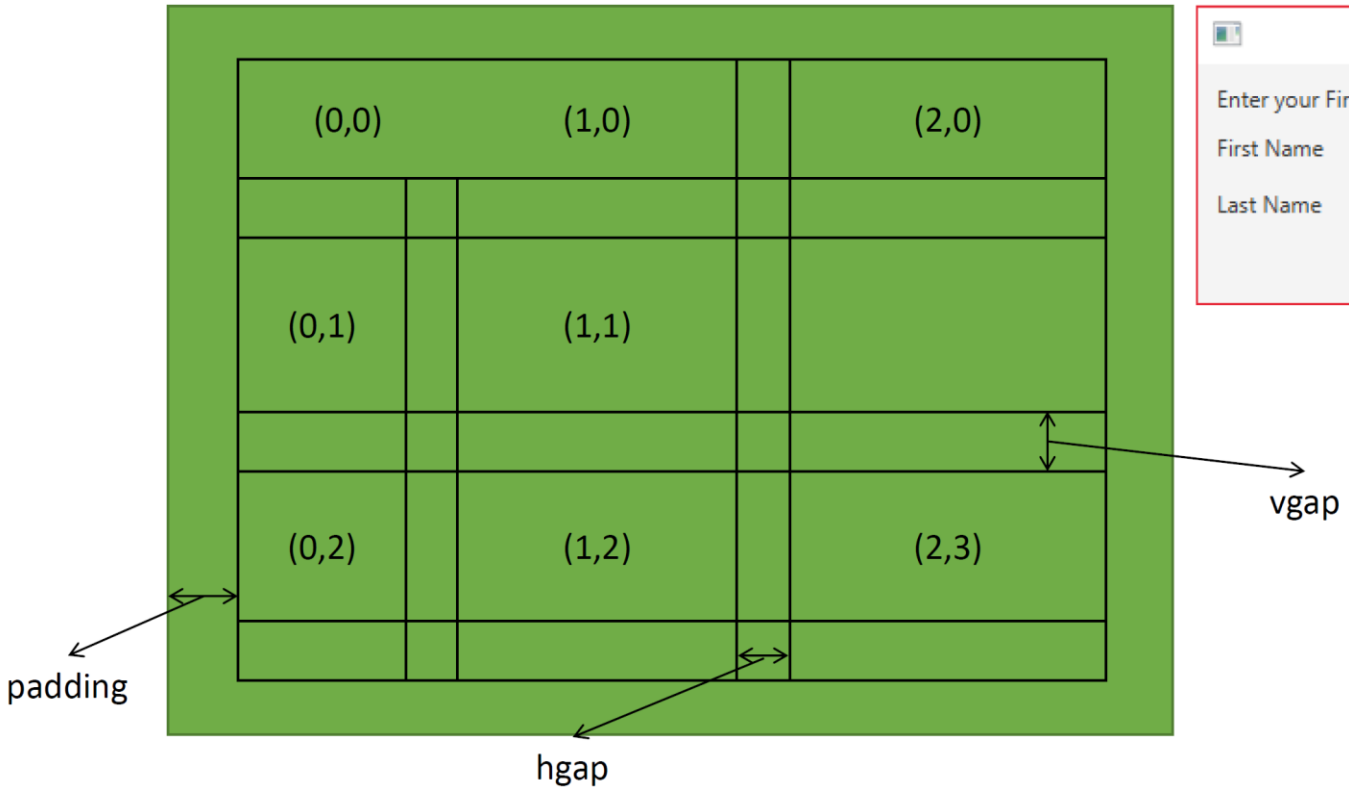
- Can also alignment of child components
  - Default positioning is in `TOP_LEFT` (Top Vertically, Left Horizontally)
  - Can change Vertical/Horizontal alignment
    - e.g. `BOTTOM_RIGHT` represents alignment on the bottom vertically, right horizontally

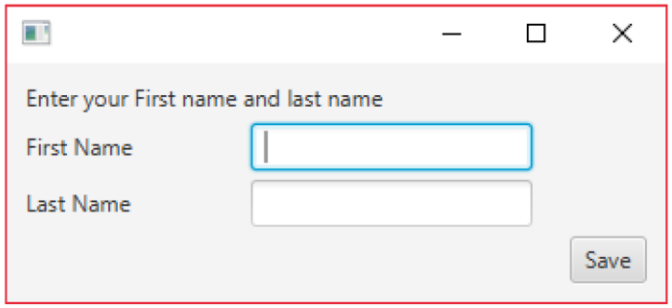


BorderPane



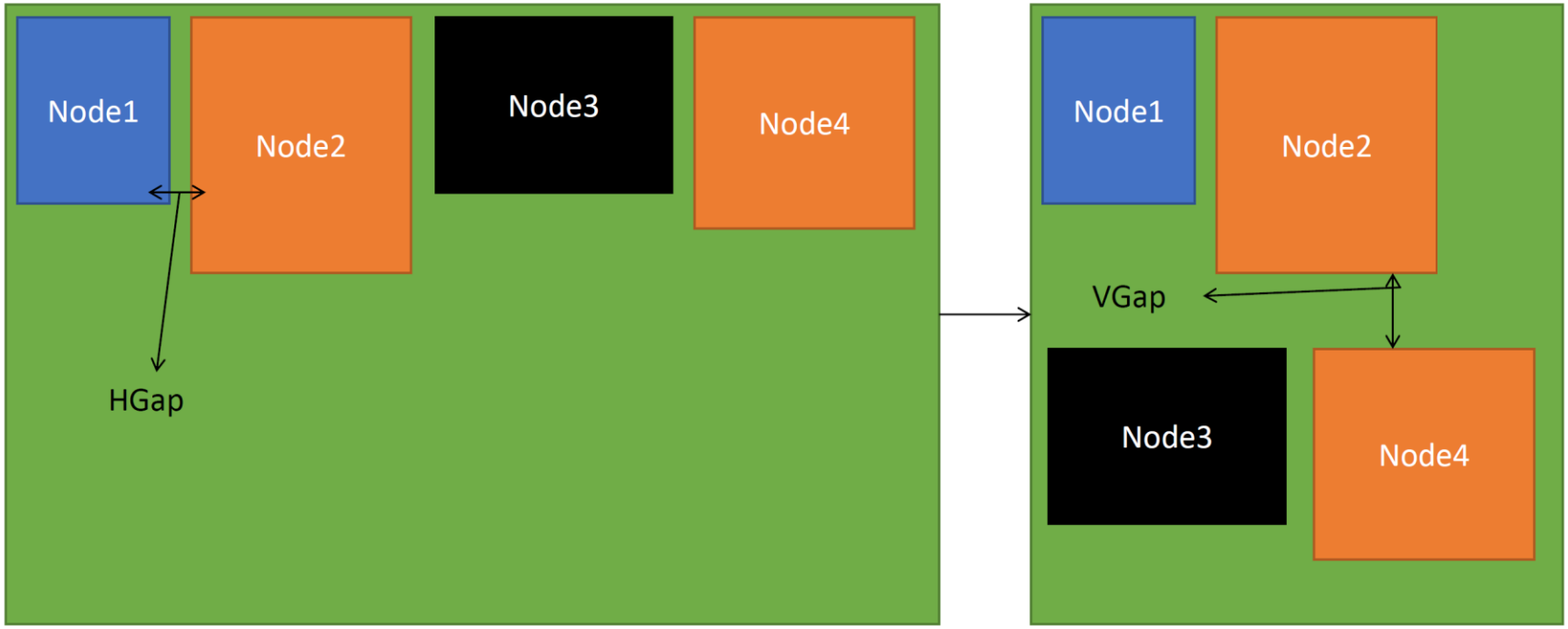
GridPane


(0,0)
(1,0)
(2,0)
(0,1)
(1,1)
(0,2)
(1,2)
(2,3)
vgap
padding
hgap

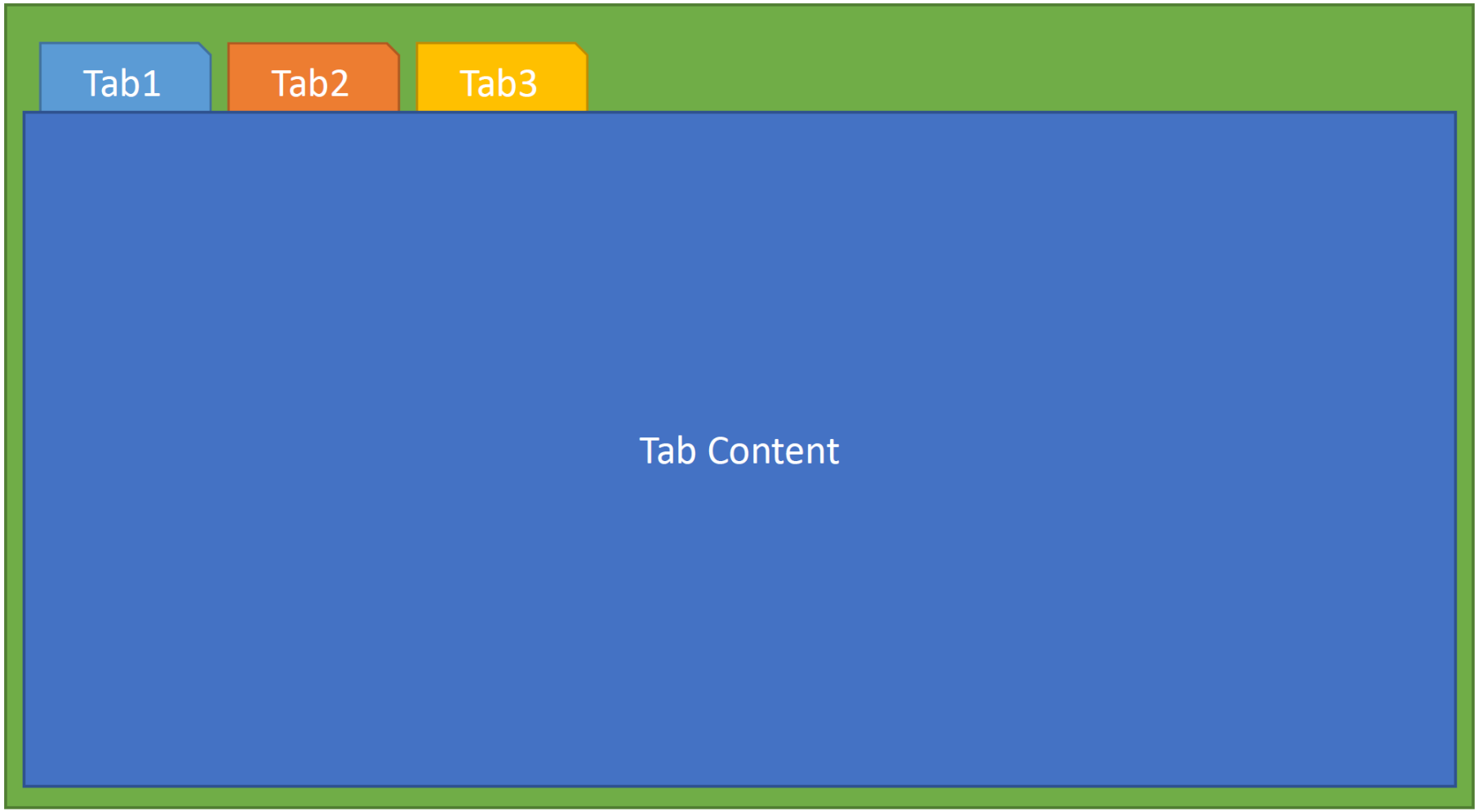

Enter your First name and last name
First Name
Last Name
Save

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