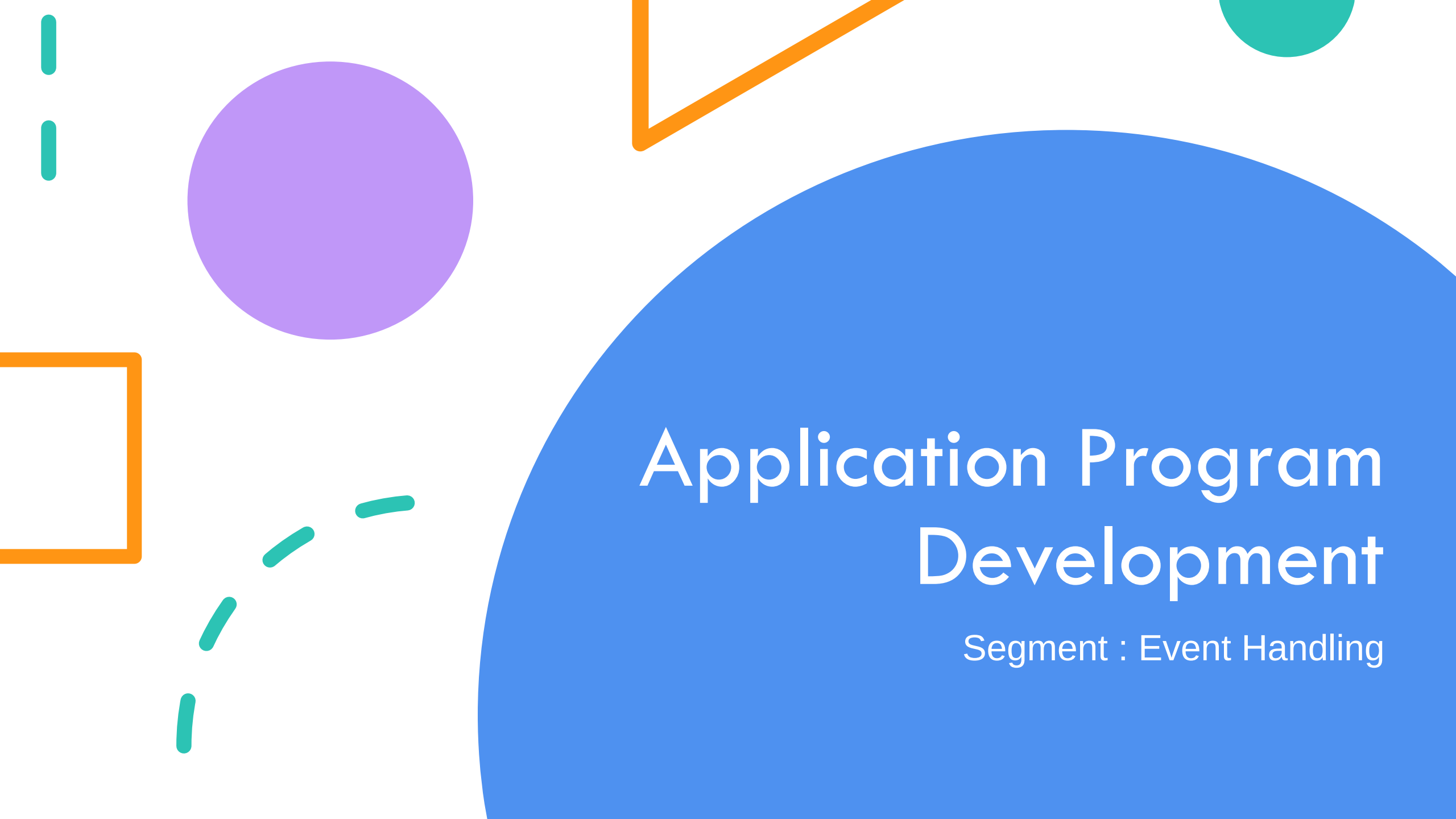

Application Program Development
Segment : Event Handling

Outcomes

- Understanding of Events in JavaFx
- Understanding of event source, target and types
- Understanding the event processing mechanism
- Understanding the event filters and handlers

What is an Event?

- The term ***event*** is used to describe an occurrence of interest.
- In a GUI application, an event is an occurrence of a user interaction with the application.
- Clicking the mouse and pressing a key on the keyboard are examples of events in a JavaFX application.

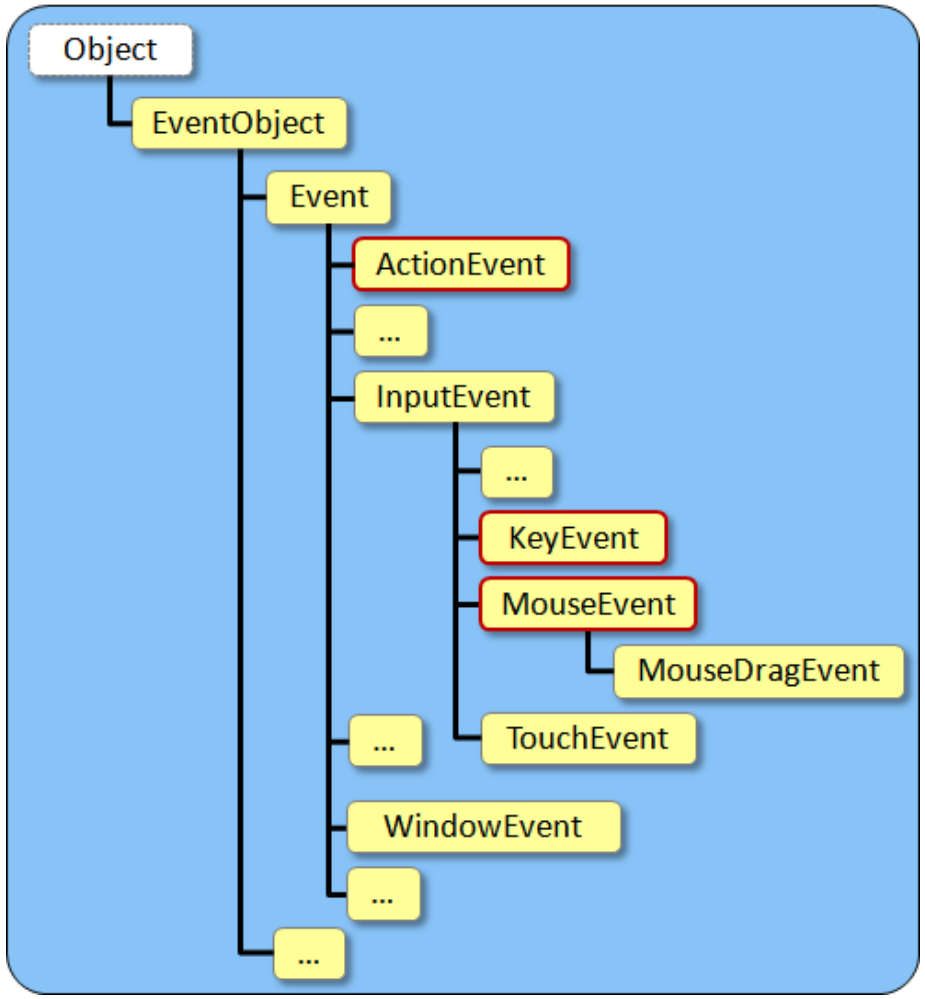
Three important definition

- *An **event** is something that happens in the program based on some kind of triggering input which is typically caused (i.e., **generated**) by user interaction such as pressing a key on the keyboard, moving the mouse, or pressing a mouse button.*

- *The **source** of an event is the component for which the event was generated (i.e., when handling button clicks, the **Button** is the **source**).*

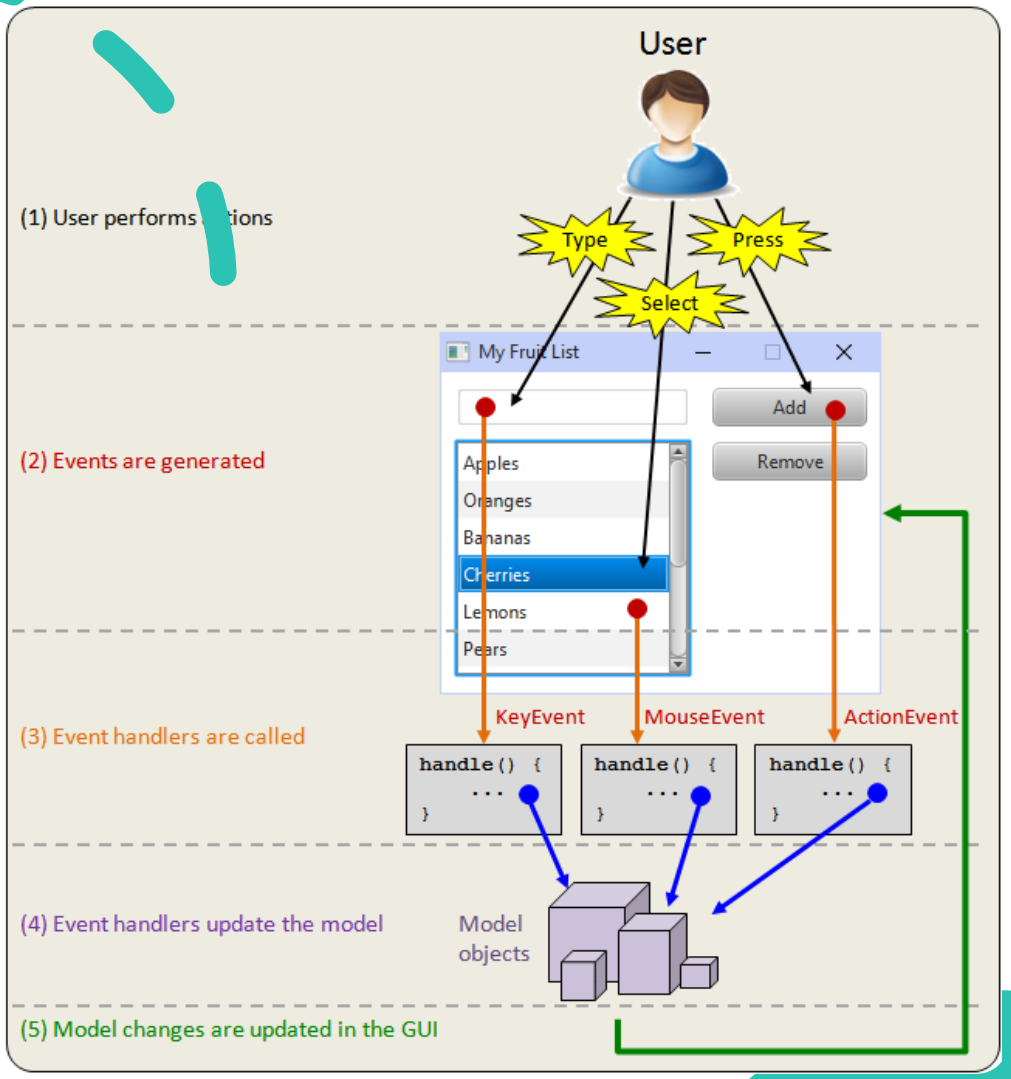
- *An **event handler** is a procedure that contains the code to be executed when a specific type of event occurs in the program.*

- In JAVA FX, all **Events** are represented by a distinct class. There are many kinds of events, each having its own unique class. Here is a partial hierarchy showing just a few of the events in the class hierarchy. We will look at the three highlighted in red



Understanding events flow

- These events are generated when the user interacts with the user interface as follows:
  - The user causes an event by clicking a button, pressing a key, selecting a list item etc..
  - Events are generated
  - The appropriate event handler is called.
  - The event handling code changes the model in some way.
  - The user interface is updated to reflect these changes in the model.



- Therefore, to perform event handling in JAVA FX, you must first identify the types of events that you want to handle.
- Then you need to write the appropriate event handlers. For each event, there is a corresponding *interface* in JAVA with a list of methods that you can write in order to handle the appropriate event in a meaningful way.
- Below is a table of three of the commonly-used events along with the list of methods that you may implement to handle the kind of event that you are interested in. For a more complete description of these (and other) events, event handlers and their methods, see the JAVA FX API specifications.

Event	Method to Implement

ActionEvent - generated when button pressed, menu item selected, enter key pressed in a text field or from a timer event	`setOnAction(new EventHandler<ActionEvent>() {`
	`  public void handle(ActionEvent actionEvent) {`
	`  // ...`
	`  }});`

KeyEvent - generated when pressing and/or releasing a key while within a component.	`setOnKeyPressed(new EventHandler<KeyEvent>() {`
	`  public void handle(KeyEvent keyEvent) {`
	`  // ...`
	`  }});`
	`setOnKeyTyped(new EventHandler<KeyEvent>() {`
	`  public void handle(KeyEvent keyEvent) {`
	`  // ...`
	`  }});`
	`setOnKeyReleased(new EventHandler<KeyEvent>() {`
	`  public void handle(KeyEvent keyEvent) {`
	`  // ...`
	`  }});`

MouseEvent - generated when pressing/releasing/clicking a mouse button, moving a mouse onto or away from a component, moving the mouse over a component, and dragging something over the component.

```
setOnMousePressed(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent mouseEvent) {
        // ...
    }});
setOnMouseClicked(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent mouseEvent) {
        // ...
    }});
setOnMouseReleased(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent mouseEvent) {
        // ...
    }});
setOnMouseEntered(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent mouseEvent) {
        // ...
    }});
setOnMouseExited(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent mouseEvent) {
        // ...
    }});
setOnMouseDragged(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent mouseEvent) {
        // ...
    }});
setOnMouseMoved(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent mouseEvent) {
        // ...
    }});
```

Handling Events

- Handling an event means executing the application logic in response to the occurrence of the event.
- Application logic is contained in the event filters and handlers, which are objects of the `EventHandler` interface

```
public interface EventHandler<T extends Event> extends EventListener
        void handle(T event);
}
```

- `EventHandler` is a generic class in `javafx.event` package.
- `EventerListener` is a marker interface** in `java.util` package.
- `handle`() method receives the reference of the event object.

**Java Marker interfaces are empty interfaces with no methods in it.

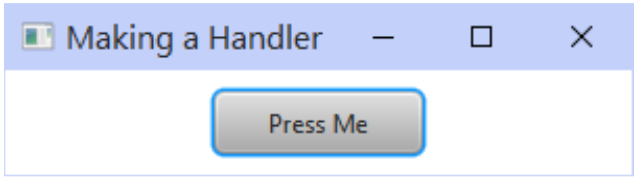
- So what does all of this mean ? It means, for example, that if you want to handle a button press in your program, you need to write a **handle()** method as follows:

```
aButton.setOnAction(new EventHandler<ActionEvent>() {
        public void handle(ActionEvent actionEvent) {
                // Write your code in here }
});
```

- Notice that we "plug-in" (or register) the event handler by calling the **setOnAction()** method for the **Button** object.
- In general, many applications can *listen for* events on the same component. So when the component event is generated, JAVA must inform everyone who is listening.
- We must therefore tell the component that we are listening for (or waiting for) an event.
- If we do not tell the component, it will not notify us when the event occurs (i.e., it will not call our event handler).
- For every event, therefore, that we want to handle, we must not only write the event handler but also **register** that event handler too.
- Imagine signing up on a webpage somewhere to receive an email notification when some event occurs (e.g., when something goes on sale, or getting an email bill-statement at the end of the month). When we sign up, we are essentially registering for (or listening to) any updates that may occur as a result of the event.

Example

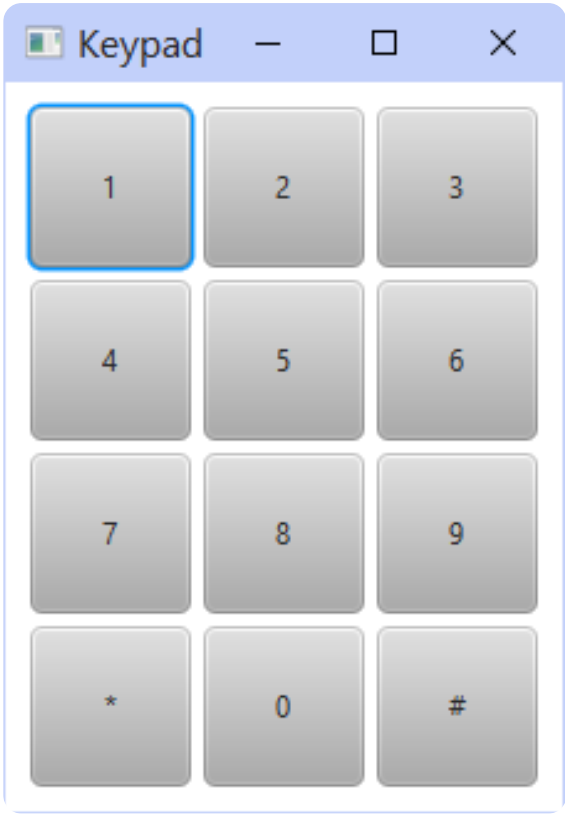
- Consider an application that will handle a simple button press.




```
import javafx.application.Application;
import javafx.event.*;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;
public class SimpleEventTest extends Application {
        public void start(Stage primaryStage) {
                Pane aPane = new Pane();
                Button aButton = new Button("Press Me");
                aButton.relocate(100, 10);
                aButton.setPrefSize(100, 30);
                aPane.getChildren().add(aButton);
                // Connect the event handler to the button
                aButton.setOnAction(new EventHandler<ActionEvent>() {
                        public void handle(ActionEvent event) {
                                System.out.println("That felt good"); }
                });
                primaryStage.setTitle("Making a Handler"); // Set title of window
                primaryStage.setScene(new Scene(aPane, 300,50)); // Set size of window
                primaryStage.show();
        }
        public static void main(String[] args) { launch(args); }
}
```

Example

- Adding the buttons to the window is easy.
- To connect their event handlers, we have two choices. We can either:
  - write separate event handlers for each button, or
  - write a single event handler for both buttons.
- However, if we have more buttons, it is often good to have them share the same event handler.
- One way to do this is to have the **Pane** implement **EventHandler<ActionEvent>** and then point all the buttons to it. We will create the following keypad using one event handler →
- The event handler will be written as a separate method (i.e., outside of the constructor), so we will need to store the buttons as instance variables (i.e., object attributes) so that we can access the buttons from both the constructor AND the event handler:



```
import javafx.application.Application;
import javafx.event.*;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;
public class MultipleButtonsApp extends Application {
        Button[][] buttons; // This will store all the Buttons
        public void start(Stage primaryStage) {
                Pane aPane = new Pane();
                buttons = new Button[4][3];
                String[] buttonLabels = {"1","2","3","4","5","6","7","8","9","*","0","#"};
                for(int row=0; row<4; row++) {
                        for (int col=0; col<3; col++) {
                                buttons[row][col] = new Button(buttonLabels[row*3+col]);
                                        buttons[row][col].relocate(10+col*70, 10+row*70);
                                buttons[row][col].setPrefSize(65,65);
                                buttons[row][col].setOnAction(new
                                        EventHandler<ActionEvent>() {
                                // This is the single event handler for all of the buttons
                                public void handle(ActionEvent actionEvent) {
                                        System.out.println("Button " +
                                         ((Button)actionEvent.getSource()).getText() + " was pressed." ); }
                                });
                        aPane.getChildren().add(buttons[row][col]);
                        }
                }
                primaryStage.setTitle("Keypad"); // Set title of window
                primaryStage.setScene(new Scene(aPane, 225, 295)); // Set size of window
                primaryStage.show();
        }
                public static void main(String[] args) { launch(args); }
}
```

Example: Calculator

- In this example, we will only be handling one event ... that of the user pressing the **Compute** button. We will need to extract the text data from the **Value** field. The **getText()** method in the **TextField** class allows us to get the text (as a **String** object) that lies in the field. We will need to convert this **String** into a number, such as a **float**, in order to perform computations with it.
- In JAVA, we can extract a **float** from a **String** by using the following strategy:

```
float x = Float.parseFloat(aString);
```

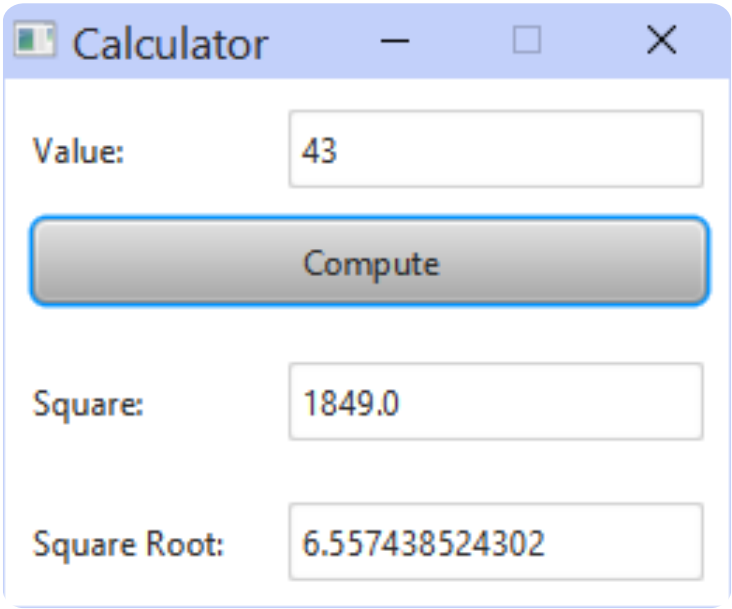
- This code will convert the **String** (called **aString** in the example) into a **float** (called **x** in the example). There are actually similar ways to convert to other types. Here are some more:

```
int x = Integer.parseInt(aString);

double x = Double.parseDouble(aString);

boolean x = Boolean.parseBoolean(aString); // "true" to true
```

- Once we have the value as a number, we can compute the square and root and then we simply need to put the results into the other two text fields. We do that using **setText()** where we supply a **String** with the result in it. The simplest way to convert a number into a String is to append the number to an empty String object in JAVA as follows:

```
aTextField.setText("" + x);
```

- This will work for any number **x**, regardless of whether it is an **int**, **float**, **double** , etc..



- One last point ... we will probably want to disable editing in the last two text fields so that the user cannot type into them, since they are "output only" fields. We use the **setEditable(false)** method to do this.

```
import javafx.application.Application;
import javafx.event.*;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;
public class CalculatorApp extends Application {
        // Text fields to hold the user data and the computed data
        TextField valueField, squareField, rootField;
        public void start(Stage primaryStage) {
                Pane aPane = new Pane();
                // Add the value label and text field
                Label label = new Label("Value:");
                label.relocate(10,10); label.setPrefSize(100, 30);
                aPane.getChildren().add(label);

                valueField = new TextField();
                valueField.relocate(100,10); valueField.setPrefSize(150, 30);
        aPane.getChildren().add(valueField);

                // Add the compute button
                Button computeButton = new Button("Compute");
                computeButton.relocate(10,50); computeButton.setPrefSize(240, 30);
                aPane.getChildren().add(computeButton);

                // Add the square label and text field
                label = new Label("Square:");
                label.relocate(10,100); label.setPrefSize(100, 30);
                aPane.getChildren().add(label);
                squareField = new TextField();
                squareField.relocate(100,100); squareField.setPrefSize(150, 30);
                squareField.setEditable(false);
                aPane.getChildren().add(squareField);
```

```
        // Add the square root label and text field
        label = new Label("Square Root:");
        label.relocate(10,150); label.setPrefSize(100, 30);
        aPane.getChildren().add(label);
        rootField = new TextField(); rootField.relocate(100,150);
        rootField.setPrefSize(150, 30);
        rootField.setEditable(false);
        aPane.getChildren().add(rootField);
        // Connect the event handlers to the buttons
        computeButton.setOnAction(new EventHandler<ActionEvent>() {
        public void handle(ActionEvent actionEvent) {
                        if (valueField.getText().length() > 0) {
                                float value = Float.parseFloat(valueField.getText());
                                squareField.setText("" + value * value);
                                rootField.setText("" + Math.sqrt(value)); }
                }
        });
        primaryStage.setTitle("Calculator");
        primaryStage.setResizable(false);
        primaryStage.setScene(new Scene(aPane, 248,178));
        primaryStage.show();
    }
    public static void main(String[] args) { launch(args); }
}
```