Objective

JavaFX Event & Event Handling

- Inner Class
- To understand the JavaFX event model
- To install action and mouse event listeners
- To accept input from buttons, text fields, and mouse
- User interface *events* include key presses, mouse moves, button clicks, and so on.

Important Note

Event handler/event listener are synonyms.

There are different ways to set up event handling with JavaFX.

Only use the methods discussed in this (and later) lectures unless you have been given permission to do so. Otherwise you may get zero for your lab assignment and exam.

They are fairly straightforward and emphasize the use of ideas that you need to be comfortable with.

- E.g. using inner classes

Event Handling

In an OS (Operating System) many events occurs. For example: user clicks mouse button, types a symbol using keyboard, or a message received.

OS runs based on events and handling events.

Modern applications use the same method.

For example, a GUI consists of many elements like button, dropbox, menu, ext.

Each of these GUI elements generates an event. For example, a button pressed, mouse moved, menu selected.

Each of these elements are source of events.

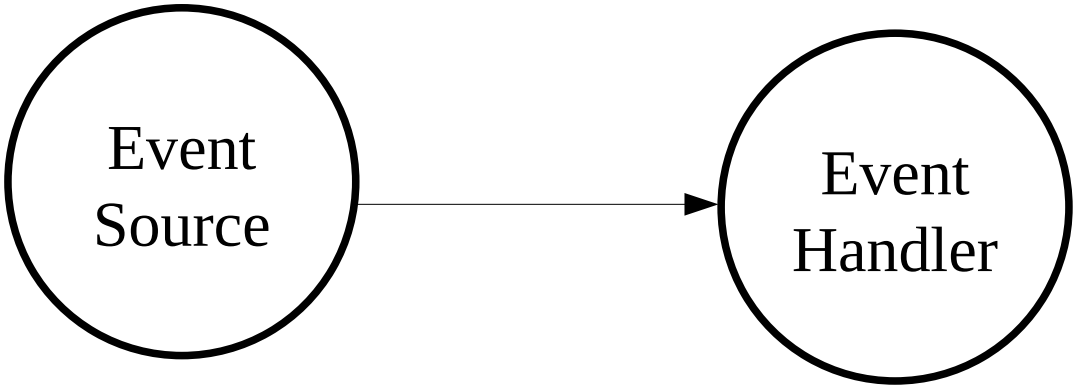
Event Handling

As the result, there are tons of events that are generated by different elements of programs.

To manage a program, every program must indicate which events it needs to receive.

A program handles the events by installing event listener objects (Agent that receives specific events).

To install a listener, you need to know the event source, then you need to add an event listener to the event source.



Events, Event Sources, and Event Listeners

Event source:

- Event sources report on events
- When an event occurs, the event source notifies event listeners

Event listener (Event Handler):

- Notified when event happens
- Its methods describe the actions to be taken when an event occurs
- A program indicates which events it needs to receive by installing event listener objects

Event

Most GUI components generate events to indicate a user action related to that component

- Ex. A button may generates an event to indicate the button was pushed

Programming technique that is oriented around events and responding to events are called event-driven programming

Event Handling

To create event and event-handling program

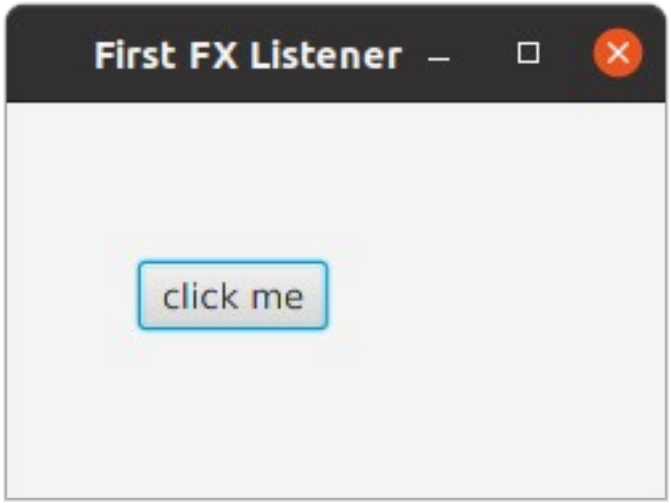
Step 1: Create event generator such as GUI elements like button

Step 2: Implement listener (event handling) class that define what will happen when particular events occur

Step 3: Set up the relationship between the listeners and the GUI elements that generate events

Example

Create a dialog with a button titled as “click me”.
When the button is being clicked, the program displays “Just clicked!” on the console (Command Window, or Terminal on Mac).



Step 1: create GUI Element

Button

The Button class creates a simple button.

The constructor allows you to set the text displayed on the button.

```
Button b = new Button(String text);
```

For now we will position the button using setLayoutX and setLayoutY

Step 1: add button

```
public class Example1 extends Application {
  private Button button;
  public void start(Stage primaryStage) {
    button = new Button("click me");
    button.setLayoutX(50);
    button.setLayoutY(60);

    Pane root = new Pane();
    root.getChildren().addAll(button);
    primaryStage.setScene(scene);
    primaryStage.show();
  }
}
```

Step 2: Implement Listener

To create an event handler, we will implement the EventHandler<T> interface

T will be a type of event. For the button, it will be ActionEvent.

There is one method to implement,

`public void handle(ActionEvent e)`

This method will be called when the event handler is triggered.

Event handlers will be inner classes, so they can interact with the data members of the outer class.

Step 2: create Listener class

```
....
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
....
private class ClickListener implements
                        EventHandler<ActionEvent> {
  @Override
  public void handle(ActionEvent e) {
    System.out.println("Just clicked!");
  }
}
```

In this simple example, we want to print message “Just clicked!” on the console, whenever the button is being clicked

Step 3: create and connect source and listener

Now we have a button and an event handler class. What happens if we click the button?

Nothing

unless, we connect the two using setOnAction method.

```
ClickListener listener = new ClickListener();
button.setOnAction(listener);
```

Refer to E1_EventFX.java

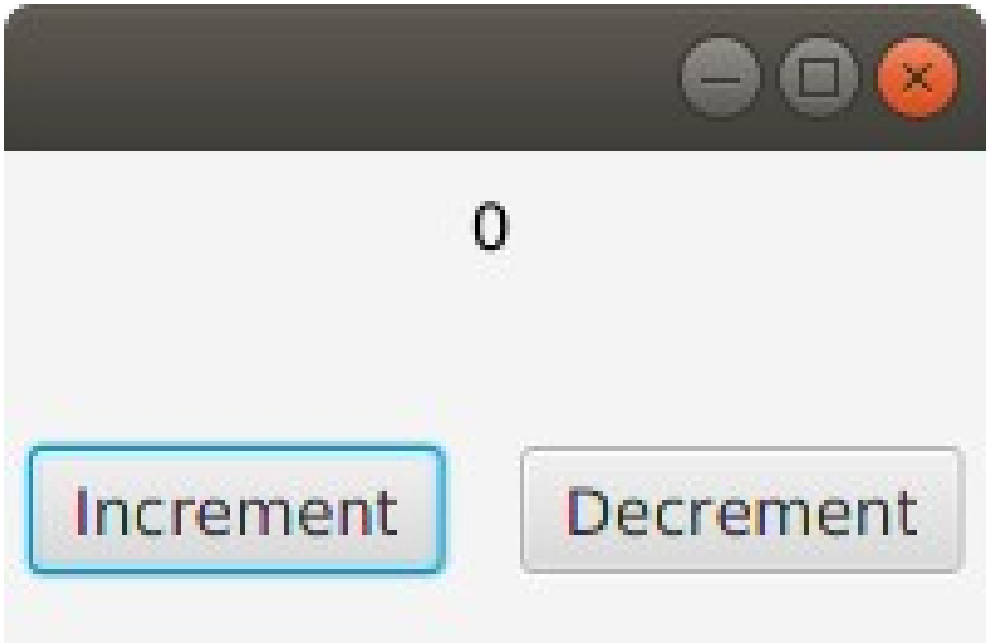
Multiple Buttons

When we are dealing with multiple buttons, we have three options:

1. Create single listener and add it to all buttons.
2. Create separate listener for each button.
3. Use combination of both item 1 and item2 based on complexity of your design.

Two Buttons

Let’s set up an example with a Text element to display a number and two buttons, one to increment, and one to decrement the number.



GUI

```
public class EventMultiButtons_A_FX extends Application {
  private Button incButton;
  private Button decButton;
  private Text valueText;
  private int val =0;
  @Override
  public void start(Stage primaryStage) {
    valueText = new Text(95,20,""+val);
    incButton = new Button("Increment");
    incButton.setLayoutX(5);
    incButton.setLayoutY(60);
    decButton = new Button("Decrement");
    decButton.setLayoutX(105);
    decButton.setLayoutY(60);
    Pane root = new Pane(valueText,incButton,decButton);
    ....
```

One Listener

We can use a single listener to handle both buttons in this case.

```
IncListener listener = new IncListener();
// bind both buttons to a single listener
incButton.setOnAction(listener);
decButton.setOnAction(listener);
```

we can use getSource() method of the ActionEvent to find out which GUI element is the source of the event.

One Listener

```
private class IncListener implement
                        EventHandler<ActionEvent> {
  @Override
  public void handle(ActionEvent e) {
    // gets source of the event
    if(e.getSource()==incButton)
      val+=1;
    else
      val-=1;
    valueText.setText(val+"");
  }
}
```

Refer to E2_EventMultiButtonsFX.java

Multiple Listeners

Alternatively, we can implement a listener class for each button separately.

```
IncListener inclistener = new IncListener();
DecListener declistener = new DecListener();
incButton.setOnAction(inclistener);
decButton.setOnAction(declistener);
```

Multiple Listeners

```
private class IncListener implements
                    EventHandler<ActionEvent> {
  @Override
  public void handle(ActionEvent e) {
    valueText.setText(++val+"");
  }
}

private class DecListener implements
                    EventHandler<ActionEvent> {
  @Override
  public void handle(ActionEvent e) {
    valueText.setText(--val+"");
  }
}
```

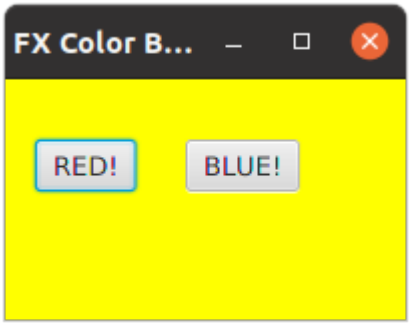
Refer to E3_EventMultiButtons2FX.java

Exercise

Make the ColorButtonsFX Application's buttons functional

- The red button turns the background rectangle red
- The blue button turns the background rectangle blue

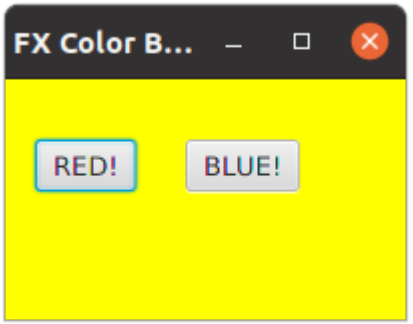
- There is starter code that sets up all needed GUI elements
- You need to create the listener class and connect it to the buttons



Exercise

```
public class ColorButtonsFX extends Application {
  private Button redButton;
  private Button blueButton;
  private Rectangle background;
  @Override
  public void start(Stage primaryStage) {
    redButton = new Button("RED!");
    redButton.setLayoutX(15);
    redButton.setLayoutY(30);
    blueButton = new Button("BLUE!");
    blueButton.setLayoutX(90);
    blueButton.setLayoutY(30);

    background = new Rectangle(0,0,200,120);
    background.setFill(Color.YELLOW);
    Pane root = new Pane();
    root.getChildren().addAll(background, redButton, blueButton);
    Scene scene = new Scene(root, 200, 120);
    primaryStage.setTitle("FX Color Buttons");
    primaryStage.setScene(scene);
    primaryStage.show();
  }
```



Refer to E4_ColorButtonsFX.java

More GUI Elements

Next we will look at other input controls

- TextField
- Checkbox
- Radio button

We will use them all in an Application that displays a message and allows the look of it to be customized

- It will start with a textbox, a button, and a text element

The lines of code positioning the elements will be omitted from the slides

- Once we get into layouts, they won't be needed

TextField

TextField is used to input single line data

Two constructors

TextField();

TextField(String initialText);

If the cursor is in the text field, the text field component will generate an action event when enter/return button is pressed

Set/Get the text from the field using the setText() and getText() methods

- Also work for the Text element

Text Fields

Lets create the following simple program.

User enters a message in the text field.

The message is shown in a large font under the text field either enter/return is pressed (when the cursor is in the text field) or the button is licked.

Note that in the slides that follows setLocationX and setLocationY statements where removed for clarification purpose since we are going to delete them later.

```
public class MessageFX1 extends Application {
  private final String initMsg="Hello, World";
  private final int fontSize = 72;
  private TextField textField;
  private Text message;
  @Override
  public void start(Stage primaryStage) {
    message = new Text(25, 175, initMsg);
    message.setFont(Font.font("Arial",fontSize));
    textField = new TextField(initMsg);
    textField.setPrefWidth(300);
    Button updateButton = new Button("Update Message");

    MessageEventHandler msgHandler = new MessageEventHandler();
    updateButton.setOnAction(msgHandler);
    textField.setOnAction(msgHandler);

    Pane root = new Pane();
    root.getChildren().addAll(textField, updateButton, message);
    // Scene and Stage code
  }
  private class MessageEventHandler implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent e) {
      message.setText(textField.getText());
  }
}
```

Refer to E5_MessageFX1.java

MessageFX1.java

Handling Events

```
public void start(Stage primaryStage) {
   ....
   MessageEventHandler msgHandler = new MessageEventHandler();
   updateButton.setOnAction(msgHandler);
   textField.setOnAction(msgHandler);
}

private class MessageEventHandler implements
                                    EventHandler<ActionEvent> {
   public void handle(ActionEvent e) {
      message.setText(textField.getText());
}
```

Checkbox

Checkboxes can be checked and unchecked

Takes no parameters or the text for its label

CheckBox();

CheckBox(String labelText);

If the checkbox is clicked, it generate an action event

Set/get whether the checkbox is checked with

setSelected(boolean) set the box selected/not selected

isSelected() returns true if the box is selected

Checkbox

Lets modify the previous example by adding a CheckBox

when checkbox is clicked, the message being updated, and it is shown in bold font.

Checkboxes

```
private CheckBox boldBox;
@Override
 public void start(Stage primaryStage) {
    ...
     boldBox = new CheckBox("Bold"); // create the box
     Pane root = new Pane();
     root.getChildren().addAll(textField, updateButton, message,boldBox);
     // Scene and Stage code
 }
 private class MessageEventHandler implements
EventHandler<ActionEvent> {
   @Override
   public void handle(ActionEvent e) {
     message.setText(textField.getText());
     FontWeight fw = FontWeight.NORMAL;
     if (boldBox.isSelected()) {  // check if it is being selected
       fw = FontWeight.BOLD;
     }
     message.setFont(Font.font("Arial",fw,fontSize));
 }
}
```

Refer to E6_MessageFX2.java

Radio Buttons

Radio buttons are similar to checkboxes, but come in sets. Only one of the buttons in the set can be selected at a time.

- Shown as circles (checkboxes are square)

Takes no parameters or the text for its label

```
RadioButton();
RadioButton(String labelText);
```

Must be added to a ToggleGroup with the other radio buttons it is associated with

```
rb.setToggleGroup(ToggleGroup tg)
```

If the radio button is clicked, it generate an action event

Set/get whether the radio button is checked with setSelected(boolean) and isSelected()

Toggle Group

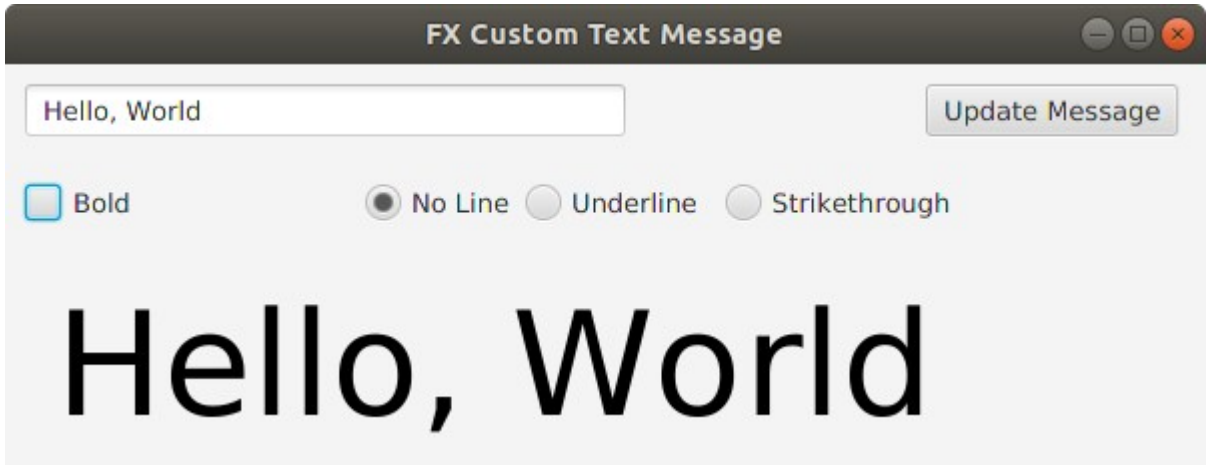
Create a ToggleGroup and add the related radio buttons to it. Otherwise they will act independently, allowing multiple radio buttons to be selected.

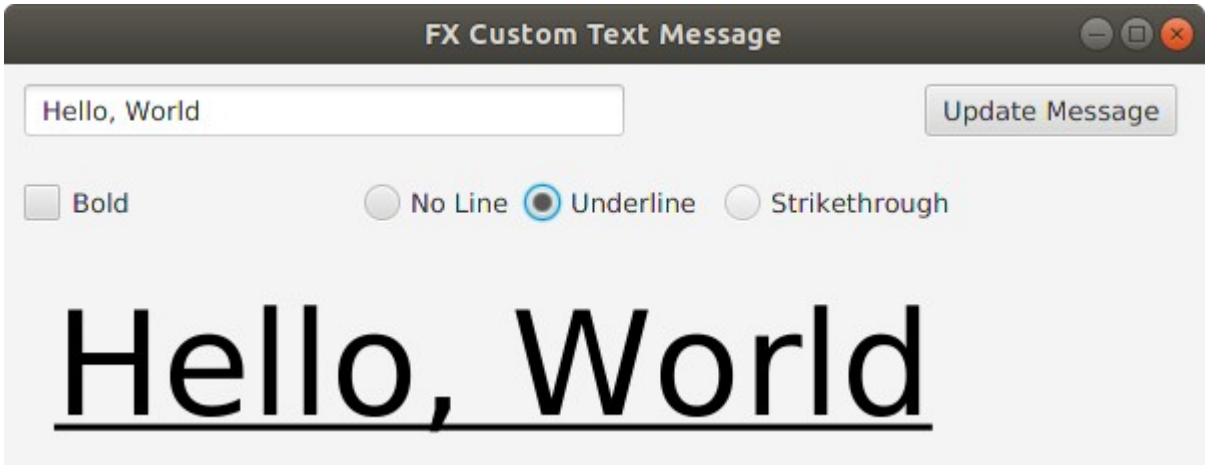
ToggleGroup();

In our example, we use isSelected to check which RadioButton is selected, but you can also use ToggleGroup's getSelectedToggle() method which returns a reference to the appropriate radio button.

MessageFX3.java

Lets modify the previous example by radio buttons.





MessageFX3.java

First we add declaration of the radiobutoons and toggleGroup to the class

```
...
private RadioButton noLineRB;
private RadioButton underlineRB;
private RadioButton strikeThroughRB;
private ToggleGroup buttonGroup;
...
public void start(Stage primaryStage) {
    ...
```

MessageFX3.java

Then we create them and add them to the GUI

```
public void start(Stage primaryStage) {
    ...
noLineRB = new RadioButton("No Line");
underlineRB = new RadioButton("Underline");
strikeThroughRB = new RadioButton("Strikethrough");

buttonGroup = new ToggleGroup();
// add them all to the toggle-group
noLineRB.setToggleGroup(buttonGroup);
underlineRB.setToggleGroup(buttonGroup);
strikeThroughRB.setToggleGroup(buttonGroup);
buttonGroup.selectToggle(noLineRB);  // set as default
```

MessageFX3.java

Modify the eventHandler and bind the radio buttons to the handler

```
public void start(Stage primaryStage) {
      ...
   noLineRB.setOnAction(msgHandler);
   underlineRB.setOnAction(msgHandler);
   strikeThroughRB.setOnAction(msgHandler);
   ...
}
private class MessageEventHandler implements
EventHandler<ActionEvent> {
  public void handle(ActionEvent e) {
    message.setStrikethrough(strikeThroughRB.isSelected());
    message.setUnderline(underlineRB.isSelected());
  }
}
```

Refer to E7_MessageFX3.java

Mouse Events

To react to mouse events (clicks/movement/etc.) attach the handler to the appropriate node

The following methods allow you to attach a handler for the desired event type

node.setOnMouseClicked(EventHandler<MouseEvent> handler)

node.setOnMousePressed(EventHandler<MouseEvent> handler)

node.setOnMouseReleaseed(EventHandler<MouseEvent> handler)

node.setOnMouseMoved(EventHandler<MouseEvent> handler)

node.setOnMouseDragged(EventHandler<MouseEvent> handler)

node.setOnMouseEntered(EventHandler<MouseEvent> handler)

node.setOnMouseExited(EventHandler<MouseEvent> handler)

Mouse Click

We will start with an example where we detect mouse clicks and add a circle to the pane for each click.

setOnMouse**xxxx**(EventHandler<MouseEvent> handler)

Method, nothing changes for other mouse even

The MouseEvent object contains the x and y coordinates of the click.

Mouse Clicks

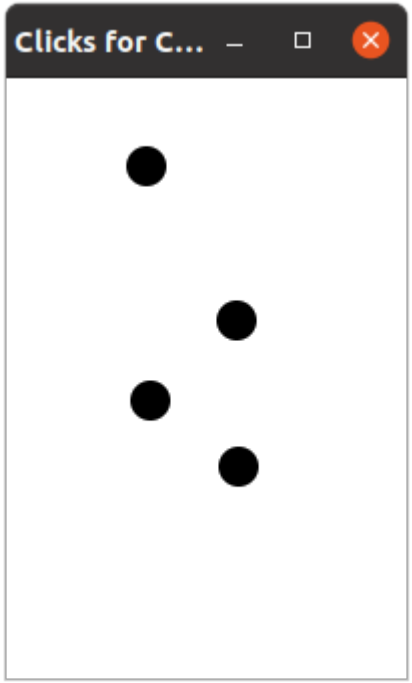
```
private Pane root;

@Override
public void start(Stage primaryStage) {
    root = new Pane();

    root.setOnMouseClicked(new ClickEventHandler());

    Scene scene = new Scene(root, 300, 400);
    primaryStage.setTitle("Clicks for Circles");
    primaryStage.setScene(scene);
    primaryStage.show();
}

private class ClickEventHandler implements EventHandler<MouseEvent> {
    @Override
    public void handle(MouseEvent e) {
        Ellipse newCircle = new Ellipse(e.getX(),e.getY(),10,10);
        root.getChildren().add(newCircle);
    }
}
```



Refer to E8_MouseClicksFX.java

Detecting Clicks

Note that we call setOnMouseClicked(...) method
we implement Eventhandler for <MouseEvent>
and we handle the (MouseEvent e) in the handle

```
public void start(Stage primaryStage) {
   root = new Pane();
   root.setOnMouseClicked(new ClickEventHandler());
}
private class ClickEventHandler implements
EventHandler<MouseEvent> {
   @Override
   public void handle(MouseEvent e) {
     Ellipse newCircle = new Ellipse(e.getX(),e.getY(),10,10);
     root.getChildren().add(newCircle);
   }
}
```

Detecting Clicks

Sometimes a single event can trigger multiple listeners.

- It will first trigger any listeners for the node that was clicked on.
- Then the listener for its parent (if present) is triggered. Then the next parent and so on to the root of the scene graph

Sometimes this is desirable, but sometimes it is not, and the event should be stopped.

Detecting Clicks

Next is an Application with a Circle. If we click the Circle, its color is set randomly. If we click anywhere else the background rectangle's color is set randomly.

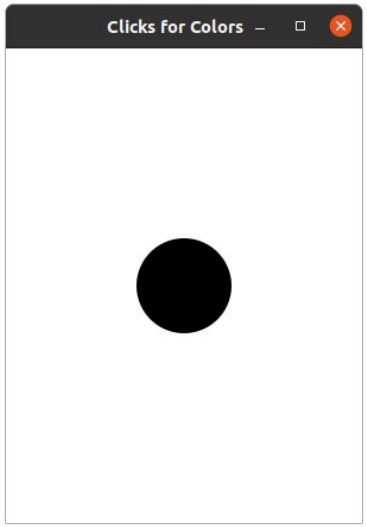
Here we have developed two event handler, one for the circle and another for the pane as shown in the next slide

```
private Pane root;
private Rectangle background;
private Ellipse centerCircle;
@Override
public void start(Stage primaryStage) {
    root = new Pane();
    background = new Rectangle(0, 0, 300, 400);
    background.setFill(Color.WHITE);

    root.setOnMouseClicked(new BGClickEventHandler());

    centerCircle = new Ellipse(150,200,40,40);
    centerCircle.setFill(Color.BLACK);
    centerCircle.setOnMouseClicked(new CircleClickEventHandler());

    root.getChildren().addAll(background, centerCircle);
    //Scene, stage code
}
```



Question

- Both event listeners are called if the circle is clicked.
  - First the circle's listener
  - Then its parent, the root's event listener
- But if the circle is clicked, we only want it to change.
- To do this, we can add one line to the circle's event listener,
  - The event is "consumed" and no longer carries on to other nodes

```
private class CircleClickEventHandler implements
EventHandler<MouseEvent> {
    @Override
    public void handle(MouseEvent e) {
       centerCircle.setFill(randomColor());
       e.consume();
    }
}
```

Detecting Clicks

```
private class BGClickEventHandler implements EventHandler<MouseEvent> {
   @Override
   public void handle(MouseEvent e) {
      background.setFill(randomColor());
   }
}
private class CircleClickEventHandler implements EventHandler<MouseEvent> {
   @Override
   public void handle(MouseEvent e) {
     centerCircle.setFill(randomColor());
     e.consume(); // we need this. why?
   }
}

private static Color randomColor() {
   Random r = new Random();
   return Color.rgb(r.nextInt(256), r.nextInt(256), r.nextInt(256));
}
```

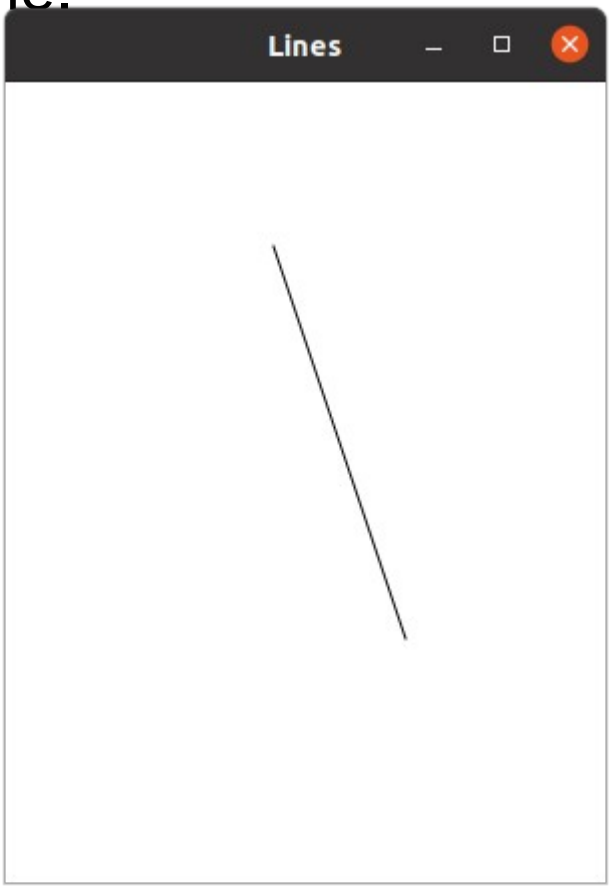
Refer to E9_ConsumeMouseEventFX.java

Mouse Drag

Next is an application to create a line with mouse click.

1. Click mouse for starting point of the line and keep the mouse pressed.

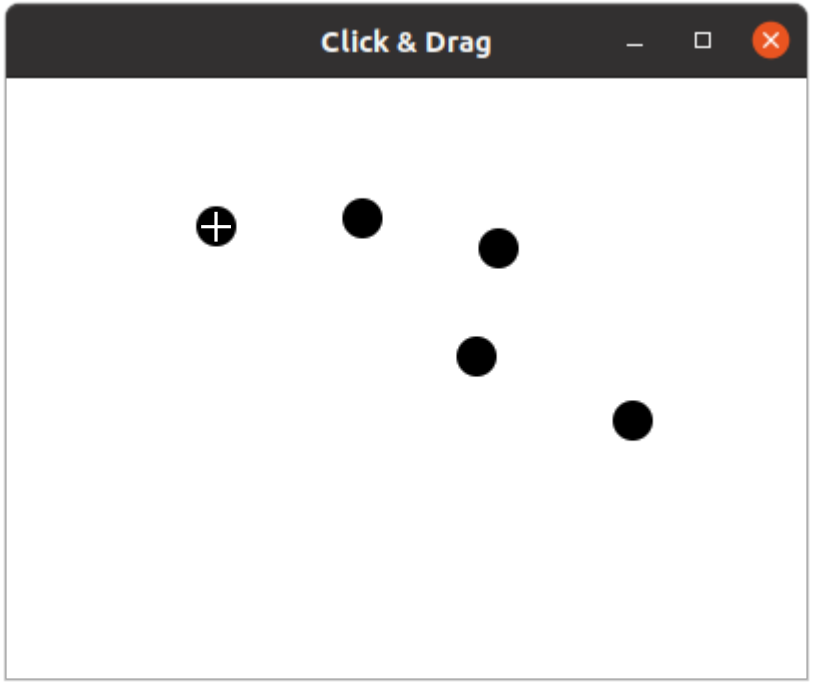
2. Drag the mouse to create the line.



Refer to E10_DragLineFX.java

Mouse Drag

Lets improve the MouseClicksFX.java. Now we want to drag the circles we have created by clicking circles and dragging them.



Refer to E11_DragCircleFX.java

Keyboard

To react to Keyboard events attach the handler to the appropriate node

The following methods allow you to attach a handler for the desired event type:

node.**setOnKeyPressed**(EventHandler<**KeyEvent**> handler)

node.**setOnKeyClicked**(EventHandler<**KeyEvent**> handler)

node.**setOnKeyReleased**(EventHandler<**KeyEvent**> handler)

Keyboard

```
Text msg = new Text(30,30,"....");
    root = new Pane(msg);
    root.setOnKeyPressed(new KeyboardPressedHandler());
    root.setOnKeyTyped(new KeyboardClickedHandler())
    ...
    stage.show();
    root.requestFocus();  // This is very important
    ......
```

Refer to KeyboardTestFX.java

Exercise

Use code provided (CircleMoveFX.java); add required event handlers to move the circle using keyboard as:

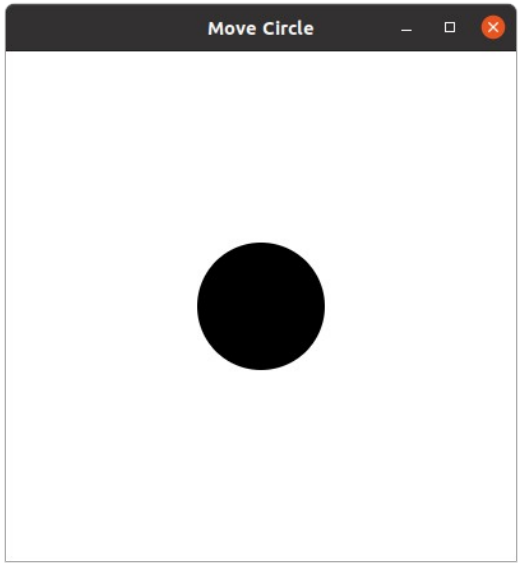
Left Key: moves the circle one pixel left

Right Key: moves the circle one pixel right

Up Key: moves the circle one pixel up

Down Key: moves the circle one pixel down

Refer to E13_MoveCircleFX.java



Summary

An inner class is declared inside another class.

Inner classes are commonly used for utilities.

Methods of an inner class can access local and instance variables from the surrounding scope.

Event source reports an event when it occurs.