



## P2. USER INTERACTION

---

Interfaces Persona Computador

Depto. Sistemas Informáticos y Computación

UPV

# Outline

- Introduction to JavaFX Events (*Event Handlers*)
- Convenience Methods
- Events and Event Handlers in FXML
- Observer Design Pattern
- Properties
- Binding

# Introduction

- Buttons and menus
- Selectors, switches, sliders, etc.
- Item selection in lists, tables, etc.
- Gestures in touch devices
- ...

What do they all have in common?

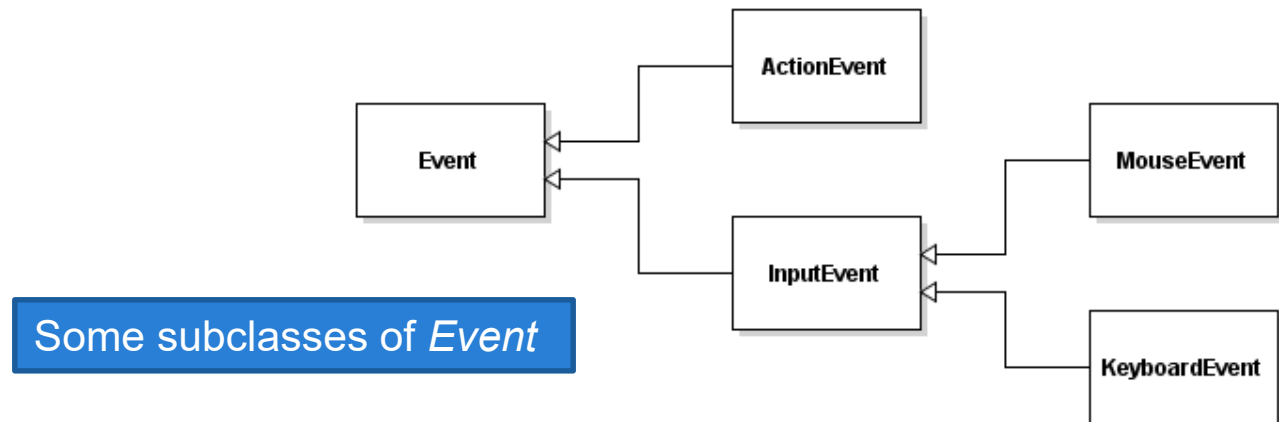
# Event-driven Interaction

- Event = notification that something happened.
  - Events are generated when the user clicks a button, presses a key, moves the mouse or performs other actions
- Event Handler: method that specify what to do when a given event is received



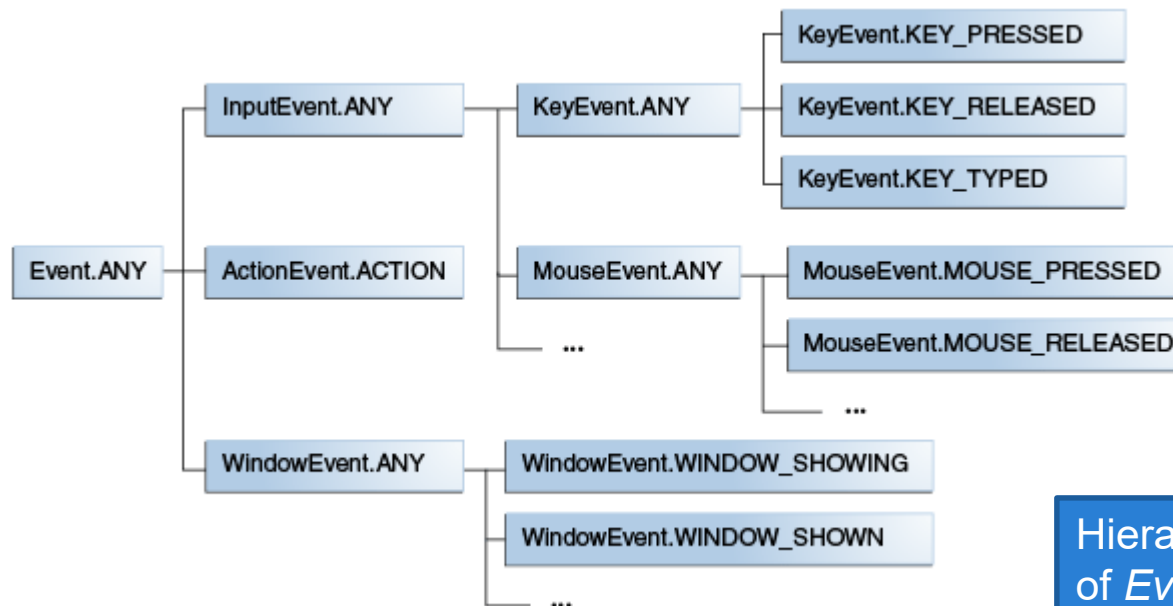
# Events

- All events inherit from the class Event
- The most important attributes of the Event class are:
  - **eventType**: an object of type EventType.
  - **source**: the object who fired the event (for example, a button can be the source of an ActionEvent, that fires whenever the button is clicked)
- The Event class has several subclasses. Each subclass has specific attributes. For example, a MouseEvent contains the window coordinates (x, y) of the position of the mouse cursor when the event was fired



# Event Types

- Each subclass of Event defines one or more instances of the class *EventType*
- The class *EventType* has an attribute called *superType*, of the same class, that defines a hierarchy of objects, as shown in the following figure



Hierarchy of instances  
of *EventType*

# Event Handlers

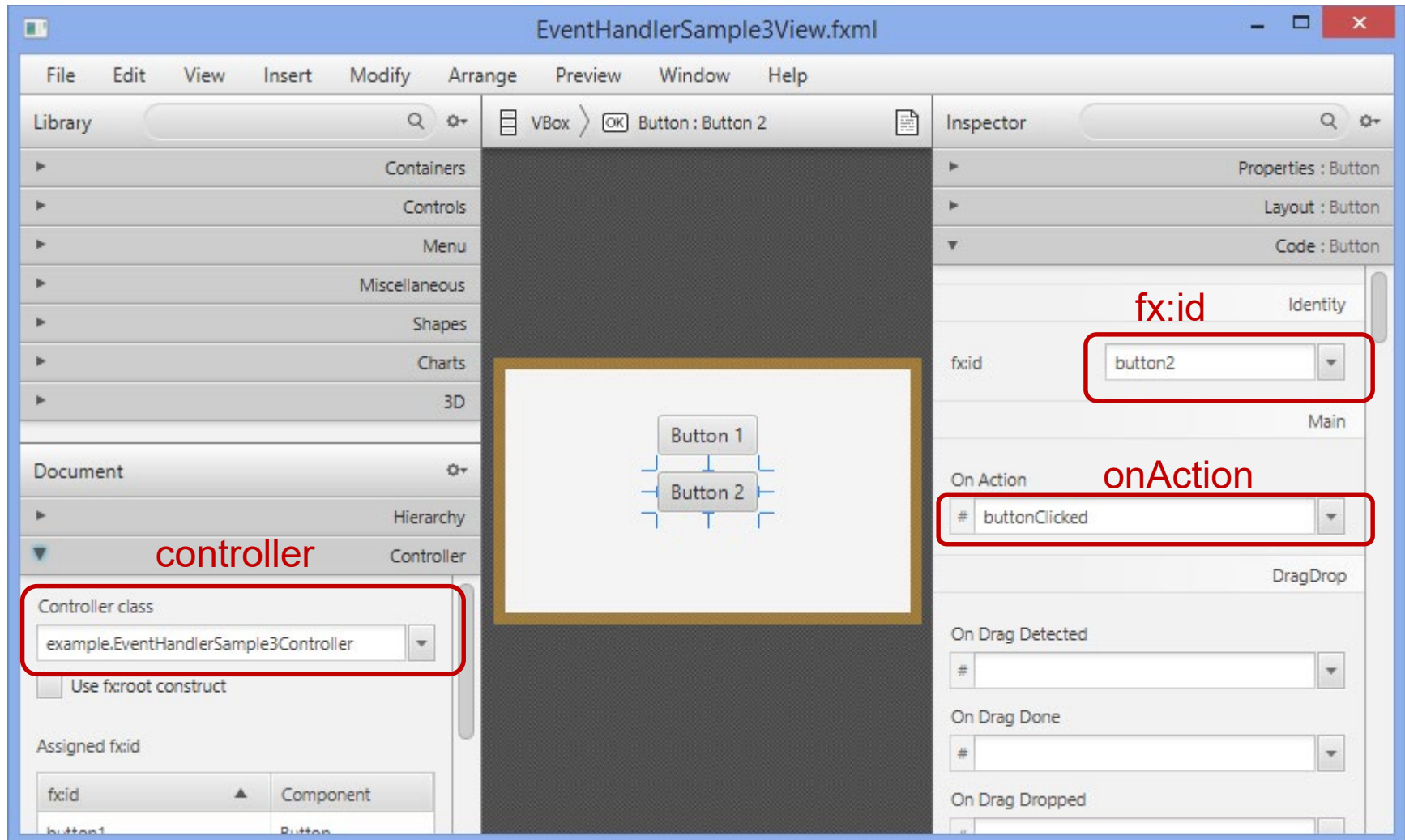
- JavaFX components generate events when the user interact with them. We can register handlers that will be notified whenever an event of a particular type is fired
  - Using the method *addEventHandler* in the *Node* class

```
void addEventHandler(EventType<T> eventType,  
    EventHandler<? super T> eventHandler)
```
  - or using a *convenience method* defined in the control class

```
setOnEventType(EventHandler<? super T> eventHandler)
```
- A handler must implement the *EventHandler<T extends Event>* interface, that declares a single method:

```
void handle(T event): method executed when a event that  
matches the desired event type is fired
```

# Adding Event Handlers in SceneBuilder: adding id and header of the handler function

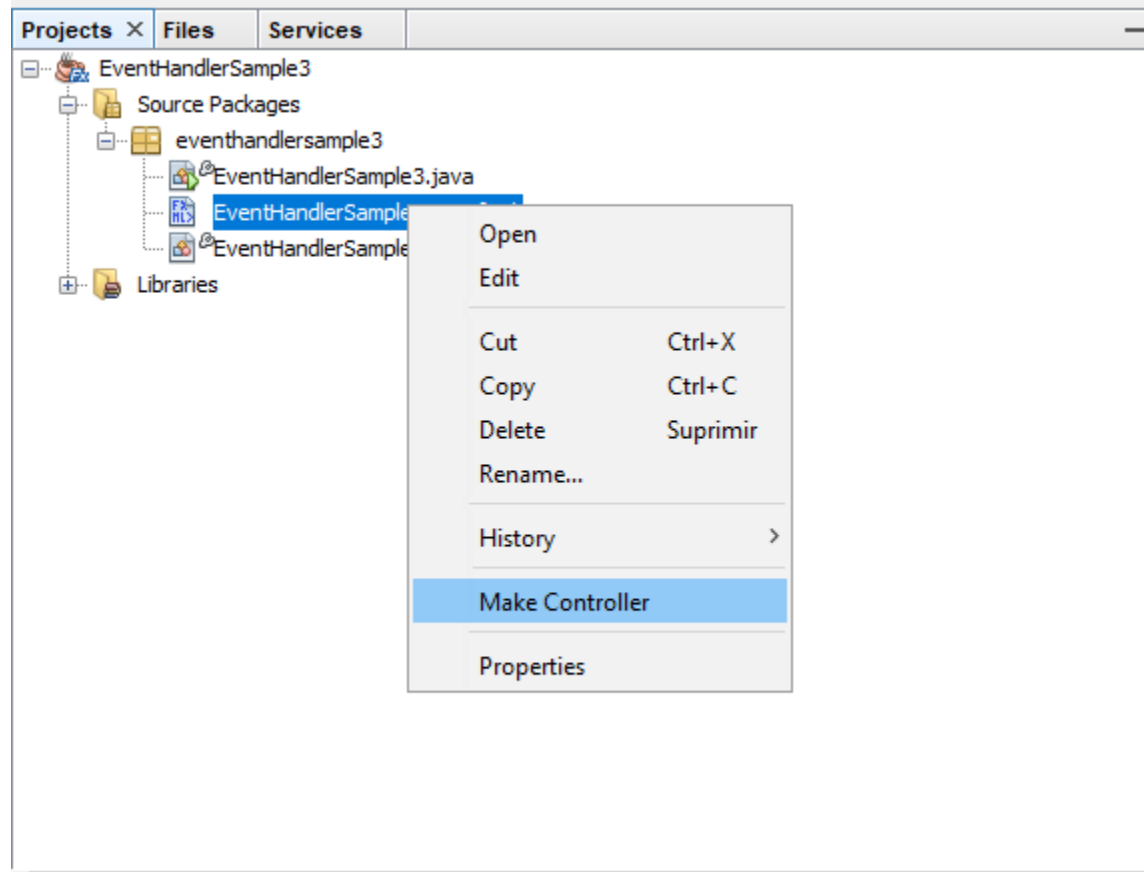




# Adding Event Handlers in SceneBuilder : FXML + Controller

```
<VBox alignment="CENTER" prefHeight="150.0" prefWidth="250.0" spacing="10.0"
xmlns="http://javafx.com/javafx/8.0.65" xmlns:fx="http://javafx.com/fxml/1"
fx:controller="example.EventHandlerSample3Controller">
    <children>
        <Button fx:id="button1" mnemonicParsing="false" onAction="#buttonClicked"
text="Button 1" />
        <Button fx:id="button2" mnemonicParsing="false" onAction="#buttonClicked"
onMouseEntered="#mouseEntered" onMouseExited="#mouseExited" text="Button 2" />
        <Label fx:id="label">
            <font>
                <Font name="Times New Roman" size="22.0" />
            </font>
        </Label>
    </children>
</VBox>
```

# Adding Event Handlers in SceneBuilder: updating view controller



# Example: FXML + Controller

```
public class EventHandlerSample3Controller {  
    @FXML  
    private Button button1;  
    @FXML  
    private Button button2;  
    @FXML  
    private Label label;  
    private static final DropShadow shadow =  
        new DropShadow();  
    @FXML  
    void buttonClicked(ActionEvent event) {  
        String id = ((Node) event.getSource()).  
            getId();  
        if (id.equals("button1")) {  
            label.setText("Button 1");  
        } else {  
            label.setText("Button 2");  
        }  
    }  
}
```

onAction

```
@FXML  
void mouseEntered(MouseEvent event) {  
    button2.setEffect(shadow);  
}  
  
@FXML  
void mouseExited(MouseEvent event) {  
    button2.setEffect(null);  
}  
}
```

# Adding Event Handlers by code

- Options:
  - Convenience Methods: short way available only for the most common events
  - *addEventHandler* method: allows adding handler to any type of event
- Exist different and equivalent ways to indicate the instructions to execute in both cases:
  - Intern classes
  - Anonym classes
  - Lambda functions -> short way, less coding
  - References to class methods

# Convenience Methods

- **ActionEvent**

- `setOnAction (EventHandler<ActionEvent> value)`

- **KeyEvent**

- `setOnKeyTyped (EventHandler<KeyEvent> value)`
  - `setOnKeyPressed (...)`
  - `setOnKeyReleased (...)`

- **MouseEvent**

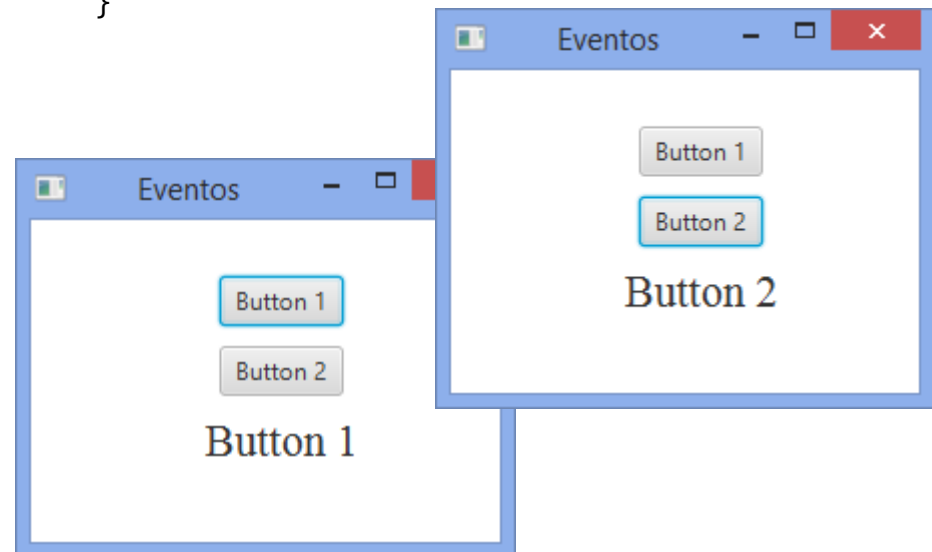
- `setOnMouseClicked (EventHandler<MouseEvent> value)`
  - `setOnMouseEntered (...)`
  - `setOnMouseExited (...)`
  - `setOnMousePressed (...)`

Learn more at: [http://docs.oracle.com/javafx/2/events/convenience\\_methods.htm](http://docs.oracle.com/javafx/2/events/convenience_methods.htm)

# View created by code. Example

```
public class EventHandlerSample1
    extends Application {
private Label label;
    @Override
public void start(Stage stage) {
    // Create the controls
    label = new Label();
    label.setFont(Font.font("Times New Roman", 22));
    Button button1 = new Button("Button 1");
    Button button2 = new Button("Button 2");
    // Create a container and insert the controls
    VBox vbox = new VBox();
    vbox.setAlignment(Pos.CENTER);
    vbox.setSpacing(10);
    vbox.getChildren().add(button1);
    vbox.getChildren().add(button2);
    vbox.getChildren().add(label);
    // Create the scene
    Scene scene = new Scene(vbox, 250, 200);
```

```
    // Setting stage properties
    stage.setTitle("Eventos");
    // Setting the scene and showing the stage
    stage.setScene(scene);
    stage.show();
}
public static void main(String[] args) {
    launch(args);
}
}
```



# Adding Handlers by code: option 1

## Inner Classes

```
@Override
public void start(Stage stage) {
    ...
    button1.addEventHandler(ActionEvent.ACTION,
        new Button1ActionHandler());
    ...
}
```

```
class Button1ActionHandler implements
    EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent event) {
        label.setText("Button 1");
    }
}
```

Adding a handler for the on  
action event, using the  
**AddEventHandler** method and  
inner classes



# Adding Handlers by code: option 2

## Anonymous Classes

```
@Override
public void start(Stage stage) {
...
button1.addEventHandler(ActionEvent.ACTION,
    new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent event) {
            label.setText("Button 1");
        }
    });
...
}
```

Adding a handler to the on  
action event, using the  
AddEventHandler method and  
an ANONYMOUS class

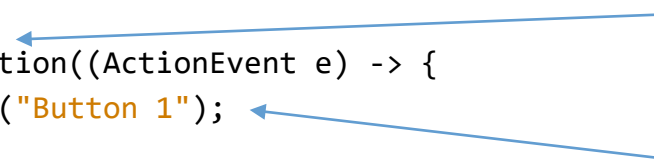


# Adding Handlers by code: option 3

## Lambda functions

```
@Override
public void start(Stage stage) {
    ...
    button1.setOnAction((ActionEvent e) -> {
        label.setText("Button 1");
    });
    ...
}
```

Adding a handler to the on  
action event, using a  
**CONVENIENCE** setOnAction  
method and a Lambda function



Lambda:

(ParameterType paramName) -> { instructions }

# Extended example

Adds the handler using  
`addEventHandler`

```
DropShadow shadow = new DropShadow();  
button2.addEventHandler(MouseEvent.MOUSE_ENTERED, (MouseEvent e) -> {  
    button2.setEffect(shadow);  
});  
  
button2.addEventHandler(MouseEvent.MOUSE_EXITED, (MouseEvent e) -> {  
    button2.setEffect(null);  
});  
  
scene.setOnKeyPressed((KeyEvent ke) -> {  
    if (ke.getCode() == KeyCode.ESCAPE) {  
        stage.close();  
    }  
});
```

Mouse events

`MouseEvent.MOUSE_ENTERED`  
`MouseEvent.MOUSE_EXITED`

Handlers  
instructions added  
using lambda  
functions

Key event handler registered using a  
convenience method  
`KeyEvent.KEY_PRESSED`

Adds the handler using a  
convenience method


# Adding Handlers by code: option 4

## Handlers instructions added using method references

```
@Override
    public void start(Stage stage) {
        ...

        button1.setId("B1");
        button2.setId("B2");
        button1.setOnAction((ActionEvent e) -> buttonClicked(e));
        button2.setOnAction(this::buttonClicked);
        ...
    }
```

They share  
the same  
handler  
method



Alternative syntax  
(method reference)

Event handler defined in a  
separated method

```
void buttonClicked(ActionEvent event) {
    Button button = (Button) event.getSource();
    String id = button.getId();
    label.setText(button.getText() + " (" + id + ")");
}
```

Using ids to identify the source  
of the event

```
void buttonClicked(ActionEvent event) {
    String id = ((Node) event.getSource()).getId();
    if (id.equals("B1")) {
        label.setText("Button 1");
    } else {
        label.setText("Button 2");
    }
}
```

# Exercise

- Create a new JavaFX FXML project
- Add to the scene graph a gridpane made of 5 rows and 5 columns
- Add a circle to the central cell of the grid.
- Add event handlers to move the circle with the arrow keys (KeyCode.UP, KeyCode.RIGHT, etc.)
- Lab work



# Exercise

- Useful Static Methods from GridPane:
  - `GridPane.getRowIndex(myNode)`: gets the row number in which the given object is displayed
  - `GridPane.getColumnIndex(myNode)`: gets the column number in which the given object is displayed
- Method to change the row or column number in which a node is displayed:
  - `miGrid.setRowIndex(myNode, rowNum)`
  - `miGrid.setColumnIndex(myNode, colNum)`
- Get the event code (i.e: which key was pressed):
  - `KeyCode.RIGHT`, `KeyCode.LEFT`, ...



# JavaBeans and Properties

- In OOP, a **property** is a means of encapsulating information with a standard interface for providing access:
  - Public access methods for reading and writing named *get/set + PropertyName*
- The Java language does not include the concept of property, but it is the foundation for the specification of JavaBeans

```
public class Node {  
    private String id;  
  
    public String getId() {  
        return id;  
    }  
    public void setId(String value) {  
        id = value;  
    }  
}
```

Example of Java class following the JavaBeans conventions

# JavaFX Properties

- A JavaFX property is a type of object that wraps or encapsulates another object (*wrapper* design pattern), adding functionality
- JavaFX classes containing properties follow the JavaBeans standard, plus a third method that returns the property (and not its value)

```
public class Node {  
    private StringProperty id = new SimpleStringProperty();  
    public String getId() {  
        return id.get();  
    }  
    public void setId(String value) {  
        id.set(value);  
    }  
    public StringProperty idProperty() {  
        return id;  
    }  
}
```

Using properties in JavaFX components

# JavaFX Properties

- We can create a property wrapping any class, but JavaFX defines property classes for all primitive types, strings and collections

`StringProperty`

`IntegerProperty`

`DoubleProperty`

`BooleanProperty`

- For generic objects JavaFX provides:

`ObjectProperty<T>`

- And for collections:

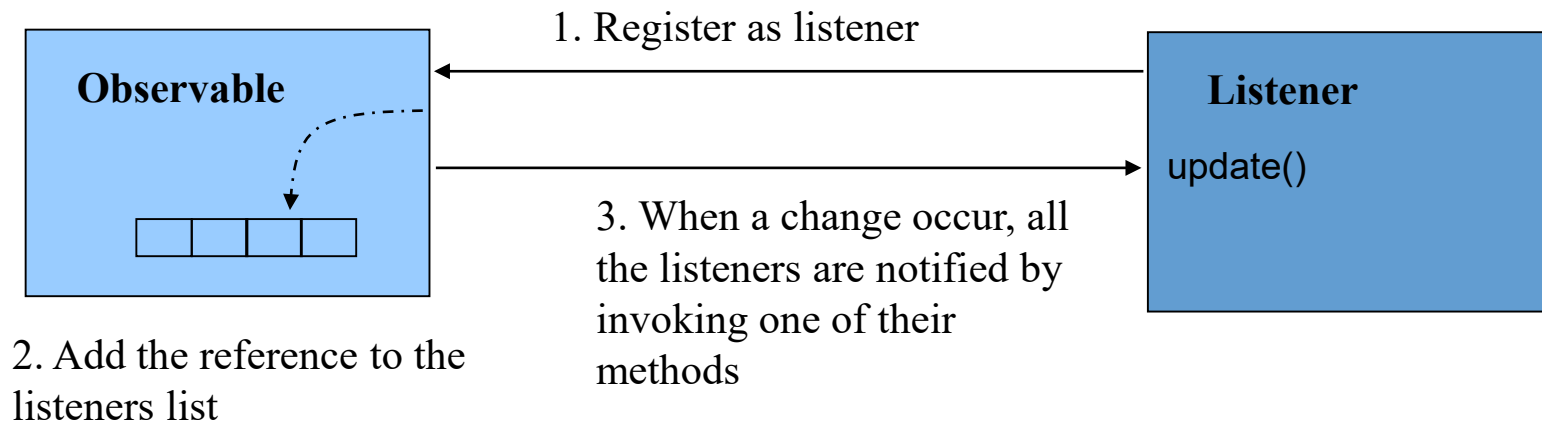
`ListProperty`

`MapProperty`



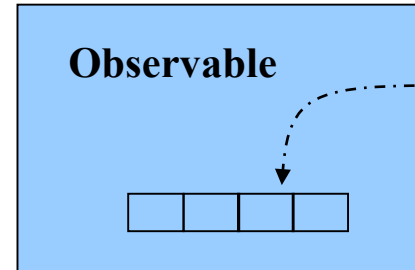
# Listening to Changes in Properties: Observer Pattern

- Dependency one-to-many between objects: when an object (source or observable object) change its state, the depending objects (listeners) are notified



# Observable objects

- Implements the ObservableValue<T> interface
  - T is type of value to be “observed”
  - The interface specify 3 methods:



`void addListener(ChangeListener<? super T> listener)`  
registers a new listener, who will be notified whenever the observed value changes

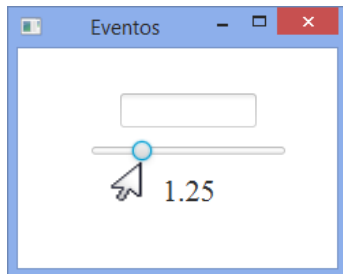
`void removeListener(ChangeListener<? super T> listener)`  
removes from the listeners list the given listener

`T getValue()` returns the current value of the observable object

# Listeners

Listener

- Implements the [ChangeListener<T>](#) interface, where T is the class of the observed value
  - Defines a single method:  
`changed(ObservableValue<? extends T> observable, T oldValue, T newValue)`
- Many JavaFX controls contain some attribute that inherits from [Property<T>](#), that inherits from [ObservableValue<T>](#), and therefore they accept listeners ([ChangeListener<T>](#))
  - We can register an object as a listener with a changed method that will be executed whenever the value of the property changes



listener

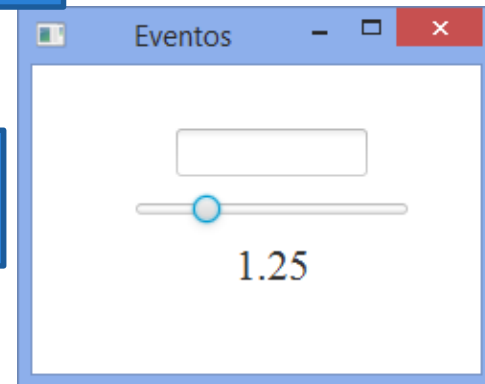
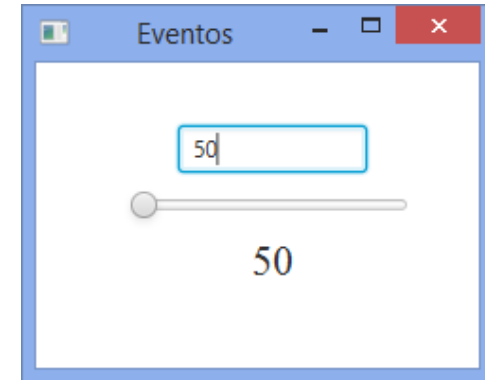
`changed(slider.valueProperty, 1, 1.25)`

# Example: TextField and Slider

```
public void initialize(URL url, ResourceBundle rb) {  
    textField.textProperty().addListener(  
        new ChangeListener<String>() {  
            @Override  
            public void changed(ObservableValue<? extends String> observable,  
                                String oldValue, String newValue) {  
                label.setText(newValue);  
            }  
        });  
    slider.valueProperty().addListener((observable, oldVal, newVal) ->  
        { label.setText(newVal + "");  
    });  
};
```

TextField listener  
(anonymous class)

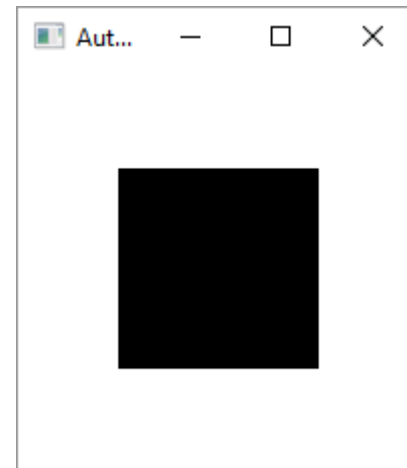
Slider listener  
(lambda function)



# Example: Elastic Rectangle

```
public void start(Stage primaryStage) {  
    Rectangle r = new Rectangle(100,100);  
    StackPane p = new StackPane();  
    p.setPrefWidth(200);  
    p.setPrefHeight(200);  
    p.getChildren().add(r);  
    p.widthProperty().addListener(  
        (observable, oldvalue, newvalue) ->  
            r.setWidth((Double)newvalue/2)  
    );  
    p.heightProperty().addListener(  
        (observable, oldvalue, newvalue) ->  
            r.setHeight((Double)newvalue/2)  
    );  
    Scene scene = new Scene(p);  
    primaryStage.setScene(scene);  
    primaryStage.setTitle("Auto Rectangle");  
    primaryStage.show();  
}
```

Listeners

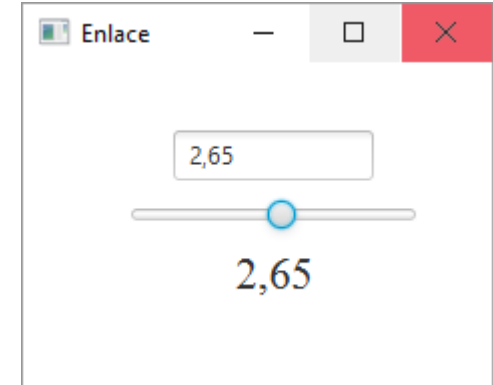


# Binding

- Unidirectional binding: if p1 is bound unidirectionally to p2, p1 will always take p2's value
  - `p1.bind(p2);`
  - Trying to manually change p1 throws an exception
- Bidirectional binding: changes in a property are propagated to the other
  - `p1.bindBidirectional(p2);`
- Bindings are created with the methods *bind*/*bindBidirectional* and deleted with *unbind*/*unbindBidirectional*.

# Example: *TextField* and *Slider*

```
Label label = new Label();  
label.setFont(Font.font("Times New Roman", 22));  
TextField textField = new TextField();  
textField.setMaxWidth(100);  
Slider slider = new Slider(0, 5, 0);  
slider.setBlockIncrement(0.5);  
slider.setMaxWidth(150);  
label.textProperty().bind(textField.textProperty());  
textField.textProperty().bindBidirectional(  
    slider.valueProperty(), new NumberStringConverter());
```



Unidirectional binding

Bidirectional  
binding

Converting from Number to String

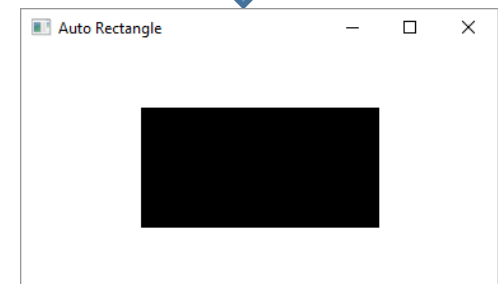
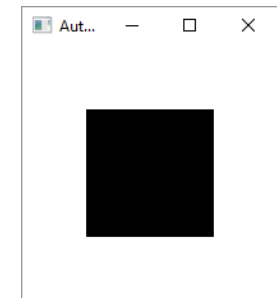
```
// label.textProperty().bind(Bindings.format("%.2f", slider.valueProperty()));
```

# Bindings

- Helper class with many utility methods: browse the documentation [here](#)

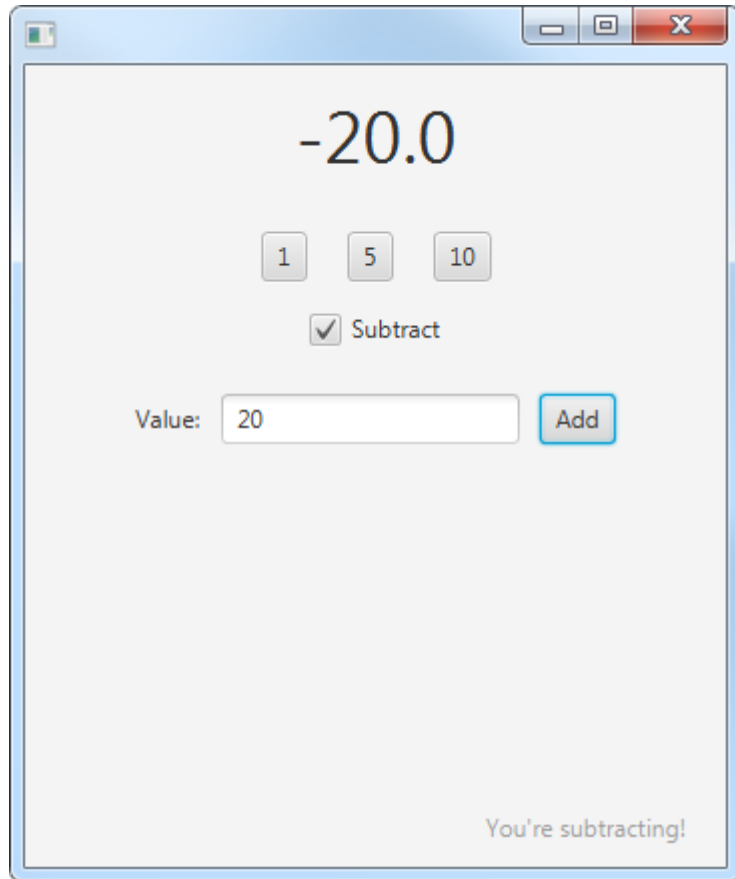
```
public void start(Stage primaryStage) {  
    Rectangle r = new Rectangle(100, 100);  
    StackPane p = new StackPane();  
    p.setPrefWidth(200);  
    p.setPrefHeight(200);  
    p.getChildren().add(r);  
    r.widthProperty().bind(  
        Bindings.divide(p.widthProperty(), 2));  
    r.heightProperty().bind(  
        Bindings.divide(p.heightProperty(), 2));  
    Scene scene = new Scene(p);  
    primaryStage.setScene(scene);  
    primaryStage.setTitle("Auto Rectangle");  
    primaryStage.show();  
}
```

Example  
Bindings.divide(...)





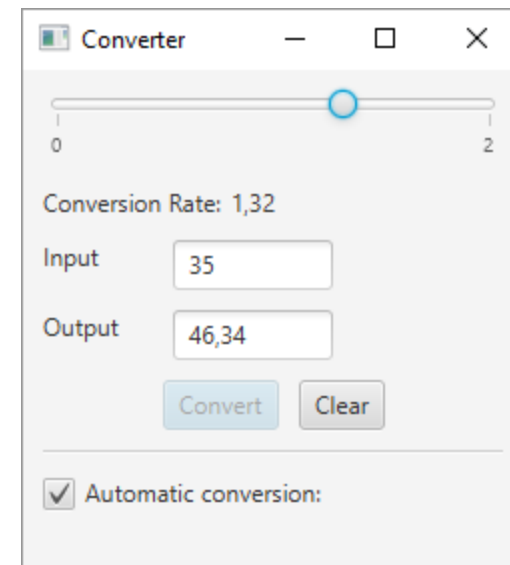
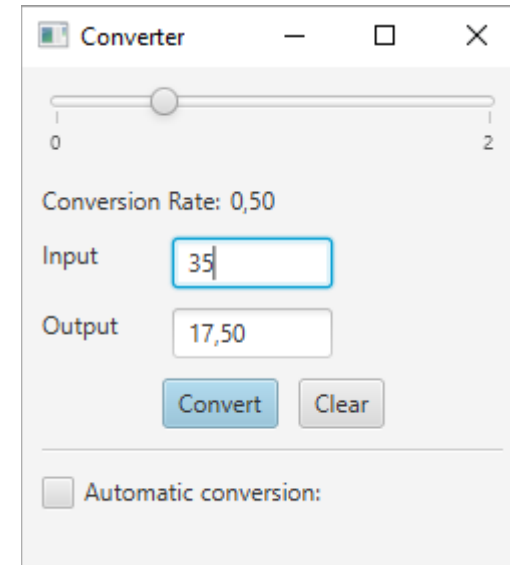
# Exercise



- Build the following application:
  - The label centered at the top is a counter
  - The buttons adds 1, 5 and 10 units to the counters.
  - The button Add sums to the counter the value shown in the TextField.
  - If the checkbox Subtract is selected, instead of adding, the previous actions subtract
  - Add a label to the bottom right corner that will only be shown when the option of subtracting is active

# Exercise

- The application multiplies an input value by a given conversion rate to get an output value.
- The conversion ratio is defined using a slider, whose value is shown below it (use **binding**)
- In the default mode, the user has to press the button *Convert* to get the result. The button *Clear* deletes both input and output values.
- In Automatic mode, the output is recomputed whenever the input value or the conversion rate change



# References

- Oracle Tutorial: Handling JavaFX Events  
<http://docs.oracle.com/javafx/2/events/jfxpub-events.htm>
- JavaFX 8 API: <https://docs.oracle.com/javase/8/javafx/api/>
- JavaFX 8 Event Handling Examples:  
<http://code.makery.ch/blog/javafx-8-event-handling-examples/>
- Lambda calculus in Java: Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft, *Java 8 in Action*  
*Lambdas, streams, and functional-style programming*