INTERACCIÓN CON EL USUARIO

Interacción Persona Computador
Depto. Sistemas Informáticos y Computación
UPV

Índice

Interacción mediante eventos

- Cuando un usuario hace clic en un botón, presiona una tecla, mueve el ratón o realiza otras acciones, se generan eventos

  Evento = notificación de que algo ha ocurrido.

- Manejador de eventos = método a ejecutar en respuesta a la ocurrencia de un evento

Eventos

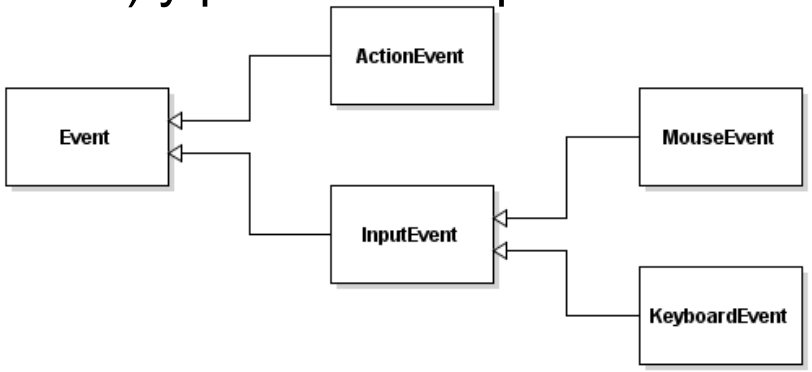
- En JavaFX cualquier evento es una instancia de la clase *Event*
  - un evento es de un Tipo determinado (*KeyEvent*, *MouseEvent, WindowEvent, ….*)
  - tiene un Source(objeto en el que ocurre la acción)
  - tiene un Target (camino a través del grafo de escena desde la raíz hasta el source // *Target* (EventDispatchChain)

- JavaFX proporciona varios mecanismos para trabajar con eventos
  - Filtros: permiten filtrar eventos a lo largo del “Target” (cadena de distribución)
  - Manejadores: métodos que especifican la respuesta a dar cuando se recibe un determinado evento

La clase Event

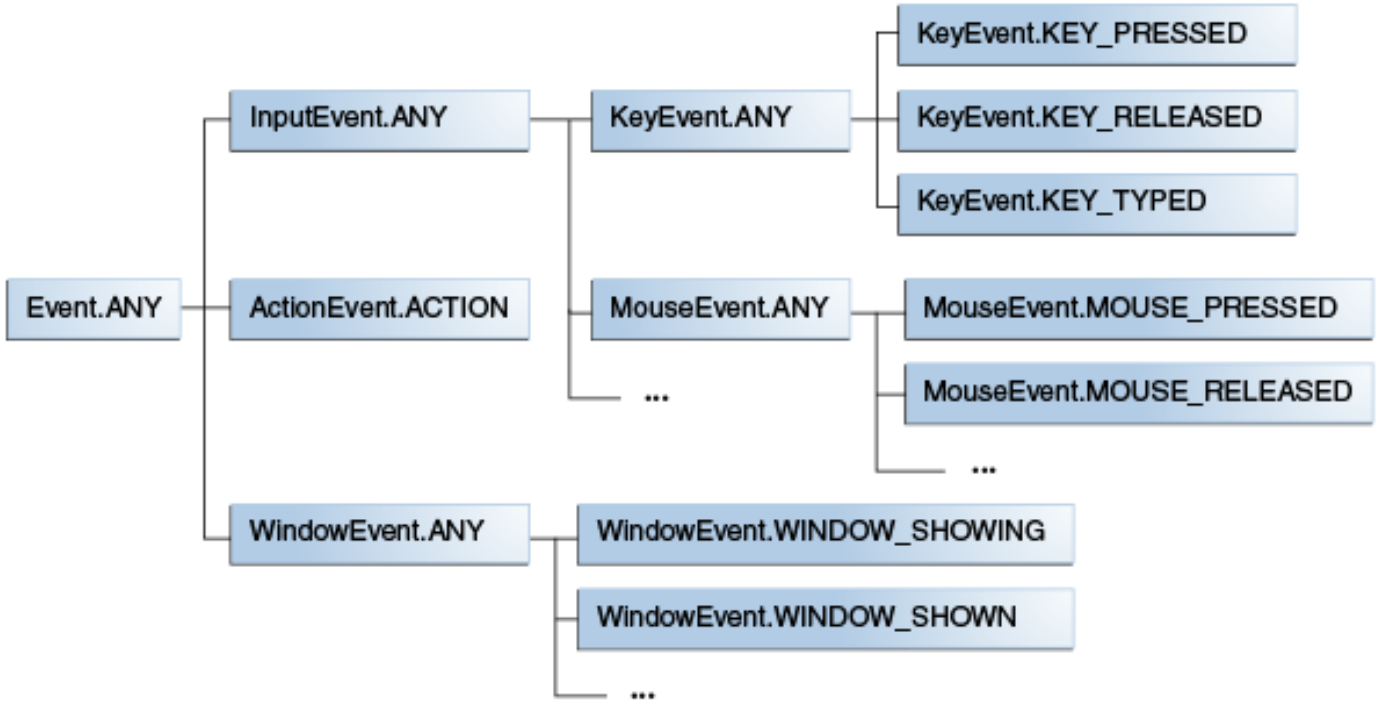
Atributo	Tipo	Descripción
eventType	EventType	Tipo de evento
*source**	Object	Origen del evento, objeto donde se produce
target	EventTarget	Describe el camino que debe seguir un evento hasta su objetivo final (EventDispatchChain).
consumed	boolean	Indica si el evento ha sido consumido por un filtro o manejador

- **Información adicional**: depende del tipo de evento, por ejemplo MouseEvent contiene, entre otras cosas, el botón pulsado, tipo de pulsación (doble, simple o mantenida) y posición del puntero del ratón.



Jerarquía de tipos de evento

Instancias de la clase **EventType**



La interfaz EventTarget

- Puede ser target de un evento cualquier clase que implemente la interfaz EventTarget

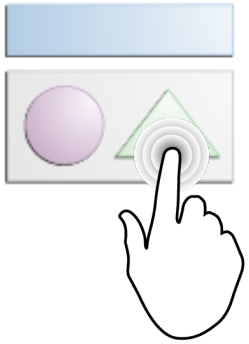
- Las clases Window, Scene y Node implementan EventTarget.
  - Cualquier subclase de ellos puede recibir eventos

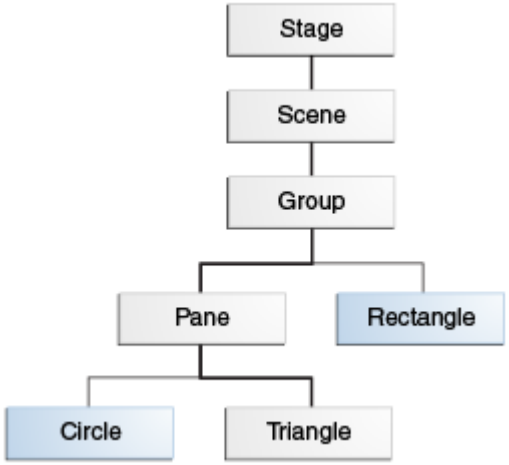
- Si creamos un componente de interfaz que no derive de Windows, Scene o Node, tendremos que implementar la interfaz EventTarget

Procesamiento de eventos (I)

1. Identificar el source:
   - Para eventos de teclado: el nodo con el foco
   - Para eventos de ratón: el nodo en la posición del puntero del ratón
   - Dispositivos táctiles: eventos de tipo gesto, *swipe* o *touch* tienen reglas más complicadas

2. Construcción de la ruta “Target”:
   - Se traza una ruta por el árbol de nodos desde el Stage al nodo objetivo.





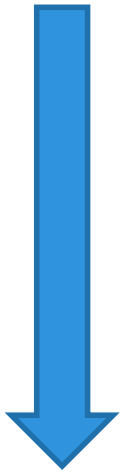
Procesamiento de eventos (II)

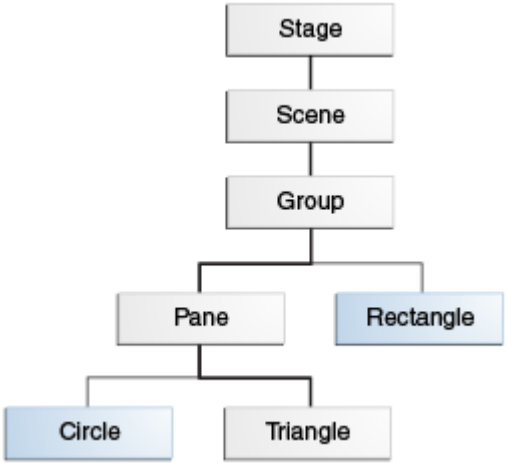
3. Captura del evento:
   - El evento recorre el la ruta desde el stage hacia el nodo source y si algún nodo por el camino tiene un **filtro** para el tipo de evento lo ejecuta.
   - Si el evento es **consumido** por algún filtro el proceso termina, en otro caso alcanzará su objetivo.

4. Propagación del evento: Tras alcanzar su objetivo, un evento vuelve al nodo raíz recorriendo la cadena en sentido inverso.
   - Algunos nodos pueden tener **manejadores** registrados para ese tipo de evento, en cuyo caso son ejecutados. Si al completar el manejador el evento no se ha **consumido**, éste sigue su recorrido hacia arriba.

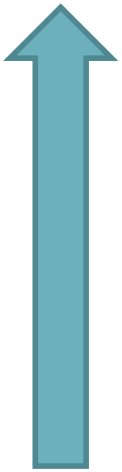
Captura y propagación

Captura del evento





Propagación



Para saber más: http://docs.oracle.com/javase/8/javafx/events-tutorial/events.htm

Codificación de los Manejadores de Eventos

- En la codificación de los manejadores tenemos dos etapas:
  1. Codificación del manejador en un método que tendrá la siguiente cabecera:

```
public void handleNombre_manejador(EventType event) {
}
```

  2. Registro del manejador sobre el objeto al que queremos añadir el manejador

- En Java 8 tenemos la posibilidad de codificar estos métodos de varias maneras:
  1. Crear una clase especifica que implemente el método (clases anónimas)
  2. Crear el método dentro de una clase y acceder al método a través del this::
  3. Utilizar expresiones lambda

La interfaz EvenHandler

Contiene SÓLO el método: public void handle(AnyEvent event)

- Es una interfaz funcional, y cualquier clase que la implemente será un manejador de eventos.
- En una asignación de este tipo podremos utilizar una expresión lambda o una referencia a un método.

```
// Definir un manejador
EventHandler handler = new EventHandler(<InputEvent>() {
    public void handle(InputEvent event) {
        System.out.println("Handling event " + event.getEventType());
        event.consume();
    }
//
```

Registro de manejadores (método 1):

- La clase Node implementa los métodos:
- addEventHandler(EventType, EventHandler)
- removeEventHandler(EventType, EventHandler)

```
// Registrar un manejador
myNode1.addEventHandler( DragEvent.DRAG_EXITED, handler );

// Desregistrar un manejador
myNode1.removeEventHandler( DragEvent.DRAG_EXITED, handler );
```

```
// Registrar un manejador CLASE ANONIMA
node.addEventHandler( DragEvent.DRAG_ENTERED, new EventHandler<DragEvent>() {
                    public void handle(DragEvent) { ... };
                });
```

Registro de manejadores (método 2):

Métodos de conveniencia

- Los métodos de conveniencia es una estrategia sencilla para registrar manejadores de eventos que se implementa en alguna clases de JavaFX
- Algunas clases de JavaFX definen propiedades del tipo manejador de evento. Al hacer un *"set"* sobre la propiedad, con un manejador definido por el usuario, se realiza automáticamente el registro del manejador con el evento asociado

```
setOnEventType(EventHandler<? super event-class> value)
```

Para saber más: http://docs.oracle.com/javase/8/javafx/events-tutorial/convenience_methods.htm#BABCCIHI

Registro de manejadores (método 2):

Métodos de conveniencia

```
……

Button btn = new Button();
btn.setLayoutX(100);
btn.setLayoutY(80);
btn.setText(”LOGIN");

// metodo de conveniencia con clase anónima
btn.setOnAction(new EventHandler<ActionEvent>() {

    public void handle(ActionEvent event) {
        System.out.println(”has clicado en LOGIN");
    }
});

// eliminar el manejador
btn.setOnAction(null);

……
```

Registro de manejadores (método 2):

Métodos de conveniencia

- Ejemplo eventos sobre el ratón:

```
final Circle circle = new Circle(radius, Color.RED);

circle.setOnMouseEntered(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent me) {
        System.out.println("Mouse entered");
    }
});

circle.setOnMouseExited(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent me) {
        System.out.println("Mouse exited");
    }
});

circle.setOnMousePressed(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent me) {
        System.out.println("Mouse pressed");
    }
});
```

Registro de manejadores (método 2):

Métodos de conveniencia

- Ejemplo eventos del teclado:

```
final TextField textBox = new TextField();
textBox.setPromptText("Write here");

textBox.setOnKeyPressed(new EventHandler<KeyEvent>() {
    public void handle(KeyEvent ke) {
        System.out.println("Key Pressed: " + ke.getText());
    }
});

textBox.setOnKeyReleased(new EventHandler<KeyEvent>() {
    public void handle(KeyEvent ke) {
        System.out.println("Key Released: " + ke.getText());
    }
});
```

Registro de manejadores (método 2):

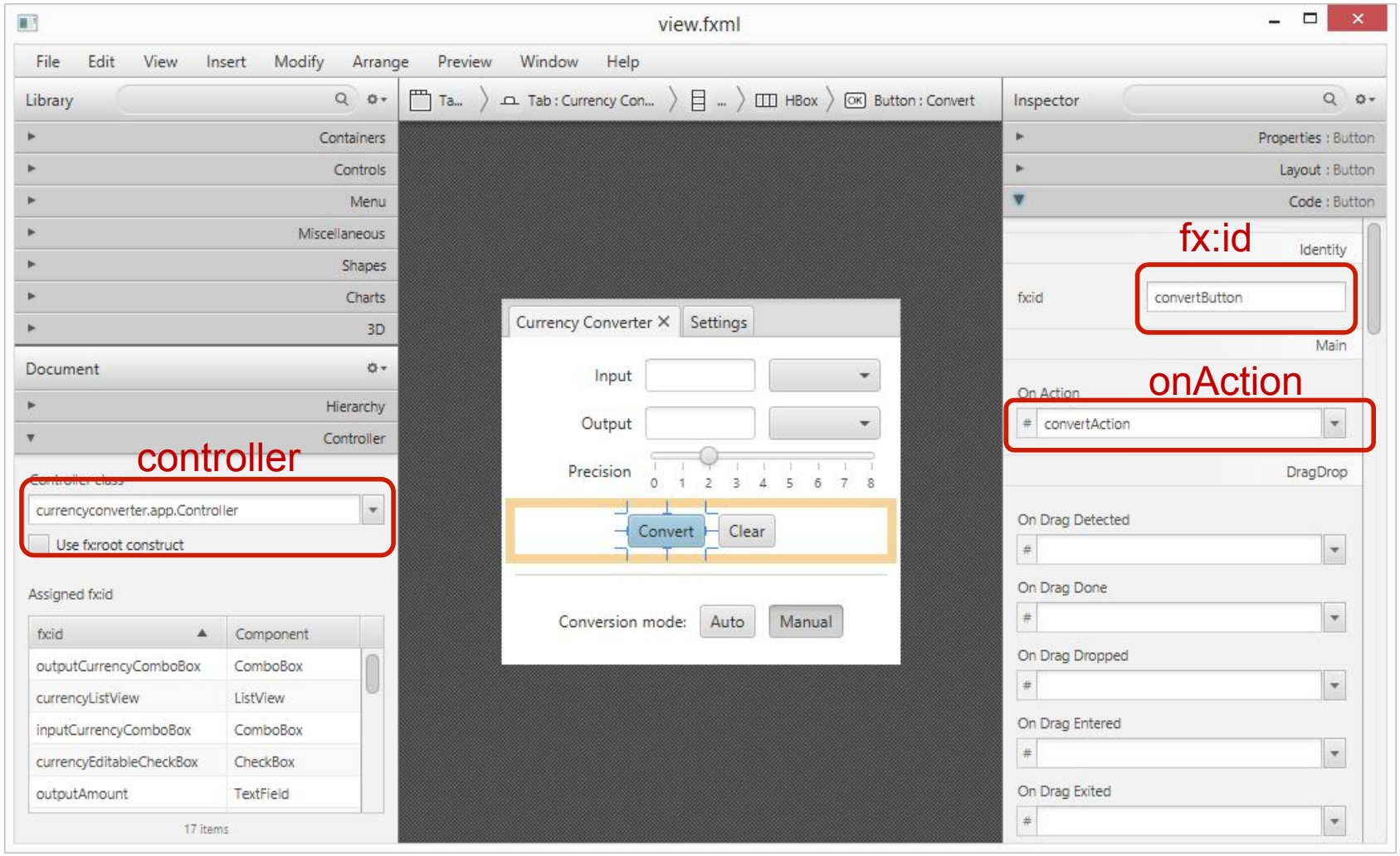
Métodos de conveniencia

- Botón JavaFX pulsado
  - `setOnAction(EventHandler<ActionEvent> value)`
- Tecla pulsada
  - `setOnKeyTyped(EventHandler<KeyEvent> value)`
  - `setOnKeyPressed(…)`
  - `setOnKeyReleased(…)`
- Botón del ratón pulsado
  - `setOnMouseClicked(EventHandler<MouseEvent> value)`
  - `setOnMouseEntered(…)`
  - `setOnMouseExited(…)`
  - `setOnMousePressed(…)`

FXML y el registro de manejadores :

- En FXML podemos asociar manejadores sobre las propiedades de manejadores de eventos. Estos manejadores deben de implementarse dentro de la clase controlador.
- ScenBuilder no mostrará para cada objeto sus propiedades de manejador de evento y tendremos que indicar el nombre del manejador.
- Si utilizamos la opción “make controller” del NetBeans los manejadores se crearán automáticamente y solo tendremos que rellenar el cuerpo de dichos manejadores
- La clase FXMLoader se encargará por nosotros de realizar las llamadas a los métodos de conveniencia para registrar los manejadores.
- En otro caso siempre es posible registrar los manejadores dentro del método initialize() de la clase controladora.

Ficheros FXML- Scene Builder



FXML

- En JavaFX, al utilizar FXML se facilita la separación de capas del patrón MVC.
- El controlador se implementa en una clase Java y se vincula al fxml mediante la declaración siguiente

```
<TabPane fx:controller="currencyconverter.app.Controller“ … >
```

- El controlador declara (etiqueta @FXML) como atributos aquellos controles de la vista con los que necesita interaccionar:

```
@FXML
private Button convertButton;
```

  - Los controles se pasan al controlador por inyección. Basta con usar en el controlador el mismo nombre dado como id en fxml.

```
<Button fx:id="convertButton" … />
```

Vista-Controlador

- En el controlador se implementan métodos para manejar los eventos de la vista, esto métodos son de tipo void y tiene como parámetro de entrada un evento.

```
@FXML
private void convertAction(ActionEvent actionEvent) {…}
```

- En fxml se establece un vínculo mediante un tag adecuado.
  - Por ejemplo, para el ActionEvent de un botón:

```
<Button fx:id="convertButton" onAction="#convertAction" … />
```

Vista-Controlador (Manejador añadido en FXML)

```
@FXML // Manejador en el controlador.
private void handleButtonAction(ActionEvent event) {
   outputTextArea.appendText("Button Action\n");}


<!-- Establecemos el manejador en la vista -->
<Button fx:id="myButton" mnemonicParsing="false" onAction="#handleButtonAction"
prefWidth="-1.0" text="Button”/
```

- Como hemos visto, los manejadores se pueden establecer desde el SceneBuilder, en la pestaña *code.*

```
public class CopyOfEventHandlingController implements Initializable {
   // Atributos que son objetos de la vista
   @FXML
   private Button myButton;
   @FXML
   private TextArea outputTextArea;
    …
@FXML  // Button event handler.
   private void handleButtonAction(ActionEvent event) {
        outputTextArea.appendText("Button Action\n");
   }
   // Método de inicialización, invocado automáticamente después de cargar el FXML
   @Override
   public void initialize() {
      }
```

Vista-controlador (Manejador añadido en initialize(), Referencias a métodos)

- Hay otra alternativa, el controlador puede implementar la interfaz *Initializable* y de esta manera se crea un método *initialize* que es invocado después de la carga del fichero FXML. Dentro de este método se realizara el registro de los manejadores de eventos.

```
public class CopyOfEventHandlingController implements Initializable {
   // Atributos que son objetos de la vista
   @FXML
   private Button myButton;
   @FXML
   private TextArea outputTextArea;

// Button event handler.
   private void handleButtonAction(ActionEvent event) {
        outputTextArea.appendText("Button Action\n");
   }
   // Método de inicialización, invocado automáticamente después de cargar el FXML
   @Override
   public void initialize() {
      // Añadir manejadores
      myButton.setOnAction(this::handlebuttonAction);
```

Vista-controlador (Manejador añadido en initialize())

- Hay otra alternativa, el controlador puede implementar la interfaz *Initializable* y de esta manera se crea un método *initialize* que es invocado después de la carga del fichero FXML. Dentro de este método se realizara el registro de los manejadores de eventos.

```
public class CopyOfEventHandlingController implements Initializable {
   // Atributos que son objetos de la vista
   @FXML
   private Button myButton;
   @FXML
   private TextArea outputTextArea;
    …
   // Otros atributos

// Método de inicialización, invocado automáticamente después de cargar el FXML
   @Override
   public void initialize() {
      // Añadir manejadores
      myButton.setOnAction(new EventHandler<ActionEvent>(){
             public void handle(ActionEvent event){
                    outputTextArea.appendText("Button Action\n");}});
…….
```

Vista-controlador (Manejador añadido en initialize(), Expresión Lambda)

- Hay otra alternativa, el controlador puede implementar la interfaz *Initializable* y de esta manera se crea un método *initialize* que es invocado después de la carga del fichero FXML. Dentro de este método se realizara el registro de los manejadores de eventos.

```
public class CopyOfEventHandlingController implements Initializable {
   // Atributos que son objetos de la vista
   @FXML
   private Button myButton;
   @FXML
   private TextArea outputTextArea;
    …
   // Otros atributos

// Método de inicialización, invocado automáticamente después de cargar el FXML
   @Override
   public void initialize() {
      // Añadir manejadores
      myButton.setOnAction((event) -> {outputTextArea.appendText("Button Action\n");});
      …
}
}
```

Ejercicio 1

- El objetivo del proyecto es trabajar con eventos del teclado.
  - Crear un proyecto JavaFX FXML nuevo (es.upv.ejercioeventos1)
  - Añadir al grafo de escena por lo menos un gridpane de 5x5 celdas.
  - Añadir un botón en el centro del grid.
  - Añadir la gestión de eventos para poder mover el botón mediante las teclas de scroll.

- El trabajo debe realizarse en el laboratorio.

JAVA 8 LAMBDAS & FXML

Cálculo lambda λ

Wiki

- Función Lambda = función anónima

  suma *S(x,y) = x + y*

  (x,y) -> x + y

- Cálculo Lambda en los lenguajes orientados a objetos implica tratar a las "funciones" como objetos de primera clase
  - Sin funciones lambda se necesita una clase para pasar un método como argumento

Interfaz funcional

- Interfaces con un único método

```
public interface Runnable {
    public abstract void run();
}
public interface EventHandler<T extends Event>{
    public abstract void handle(T event);
}
public interface ChangeListener<T>{
    public abstract void changed(ObservableValue<? extends
T> observable, T oldValue, T newValue));
}
```

- En vez de crear y pasar un objeto se pasa una lambda que implementa el método de la interfaz funcional

Ejemplos en JavaFX

- Sin funciones Lambda

```
// Handle Button event.
myButton.setOnAction(new EventHandler<ActionEvent>() {
  public void handle(ActionEvent arg0) {
    outputTextArea.appendText("Button Action\n");
  }
});
```

- Con funciones Lambda (Java 8)

```
// Handle Button event.
myButton.setOnAction((event) -> {
  outputTextArea.appendText("Button Action\n");
});
```

Java Beans

- Java Beans es el modelo de componentes de Java, donde en cada clase un atributo tienen dos métodos: uno para obtener el valor del mismo (getter) y otros para modificarlo (setter).

```
public class Person {
    private String firstName;
    private String lastName;

    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

Java Beans

- Se sigue una convención de nombres para los getters:

```
public String getLastName()
```

- De igual forma se nombran los setters:

```
public void setLastName(String lastName)
```

- Los métodos anteriores sirven para manipular propiedades de las clases.

- Las propiedades en JavaFX tienen ahora tres métodos (frente a los dos de JavaBean). Estos son:

final “getter”
final “setter” (opcional)
“property getter”

Propiedades

- Las propiedades son una nueva característica de las clases en JavaFX. Dada la clase:

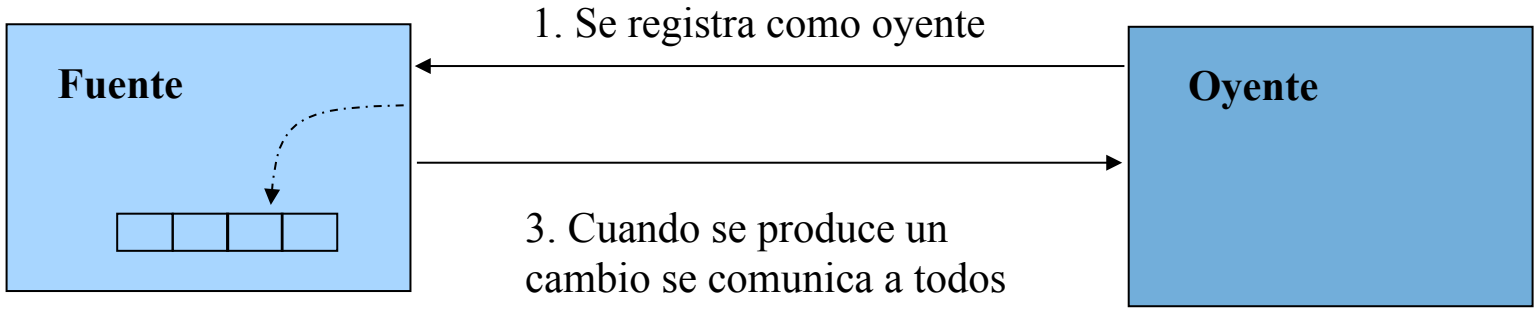
```
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;

class Factura {
          // Define a variable to store the property
          private DoubleProperty cantidad = new SimpleDoubleProperty();
          // Define a getter for the property's value
          public final double getCantidad(){return cantidad.get();}
          // Define a setter for the property's value
          public final void setCantidad(double value){cantidad.set(value);}
          // Define a getter for the property itself
          public DoubleProperty cantidadProperty() {return cantidad;}
}
```

- El tipo de los objetos no es ninguno de los tipos primitivos de Java, son una nueva clase envoltorio que encapsula al tipo primitivo y añade funcionalidad extra.

Oyentes (Listeners)

observer pattern




```
changed(ObservableValue<? extends T> observable,
T oldValue, T newValue)
```

Oyentes en JavaFX

- Se define la interfaz ObservableValue<T>
  - T es el tipo de valor que se quiere monitorizar
  - Especifica 3 métodos: *addListener(ChangeListener)*, *removeListener (ChangeListener)* y *getValue()*
- Muchos controles JavaFX contienen algún atributo de tipo Property<T>, derivado de ObservableValue<T>, y por tanto permiten añadir un ChangeListener
- Ejemplos
  - TextField.textProperty() -> StringProperty (Property<String>)
  - ListView.getSelectionModel.selectedItemProperty -> ObjectProperty<O>
  - Slider.valueProperty() ->NumberProperty(Property<Number>)

Interfaz ObservableValue<T>

Método	Descripción
void addListener(ChangeListener<? super T> listener)	Añade un ChangeListener que será informado cuando cambie el valor.
T getValue()	Devuelve al valor actual
void removeListener(ChangeListener<? super T> listener)	Borra el "listener" proporcionado de la lista de oyentes registrados para que no reciba actualizaciones de valor

Interfaz ChangeListener<T>

Método	Descripción
void changed(ObservableValue<? extends T> observable, T oldValue, T newValue)	Añade un ChangeListener que será informado cuando cambie el valor.

Referencias a métodos

```
@FXML  // Button event handler.
private void handleButtonAction(ActionEvent event) {
   outputTextArea.appendText("Button Action\n");
 }
@FXML // Slyder change listener.
private void handleSlyderChange(ObservableValue<? Extends
        Number> oValue, Number oldValue, Number newValue) {
 outputTextArea.appendText("Slider Value Changed (newValue: " +
        newValue.intValue() + ")\n");
 }
…

private void initialize() {
  myButton.setOnAction(this::handleButtonAction);
//myButton.addEventHandler(ActionEvent,this::handleButtonAction);
  mySlider.valueProperty().addListener(this::handleSliderChange);
  …
}
```

Expresiones lambda

…

```
private void initialize() {
// Button event handler
  myButton.setOnAction((e) -> {
            outputTextArea.appendText("Button Action\n");});
// Slyder change listener.
  mySlider.valueProperty().addListener( (oValue, oldValue,
            newValue) -> {
              outputTextArea.appendText("Slider Value Changed
              (newValue: " + newValue.intValue() + ")\n");
            });
}
```

Propiedades y eventos

- Puede añadirse código oyente que será informado cuando el valor de una propiedad cambia.

```
import javafx.beans.value.ObservableValue;
import javafx.beans.value.ChangeListener;

public class Principal {

@SuppressWarnings("unchecked")
public static void main(String[] args) {
// TODO Auto-generated method stub
Factura facturaElectrica = new Factura();
        facturaElectrica.cantidadProperty().addListener(new ChangeListener(){
        @Override public void changed(ObservableValue o,Object oldVal,
                Object newVal){
             System.out.println("La factura electrica ha cambiado!");
        }
      });

      facturaElectrica.setCantidad(100.00);
}

}
```

Ejercicio 1- (Ampliación 1)

- Sobre el ejercicio uno:
  1. Modificar el estilo del Stage mediante el metodo ***InitialStyle*** y construir un stage sin marco.

  2. Añadir los eventos necesarios para realizar el Drag de la ventana, utilizad expresiones lambda:
     - Paso1: añadir un manejador de eventos para el tipo MousePressed para almacenar el offset de la posición del ratón en la pantalla y la posición de la ventana en la pantalla. Se recomienda utilizar dos variables declaradas como final del tipo DoubleProperty

     - Paso 2: añadir un manejador de eventos para el tipo de evento MouseDraged. Mover la ventana en la pantalla mediante los métodos setX() y setY()

  3. Para finalizar añadir un botón en el fichero xml fuera del GridPane. Este botón permitirá cerrar la ventana invocando al metodo ***Hide*** de la clase scene. La declaración del manejador se debe de realizar en el método ***initialize*** del controlador y mediante una expresión lambda.

- El trabajo debe realizarse en el laboratorio.

Ejercicio 1 – (Ampliación 2)

1. Sobre el ejercicio uno:
   1. Añadir al controlador dos propiedades del tipo ***IntegerProperty*** para almacenar la posición del botón en el GridPane.
   2. Modificar el código en el controlador para que la posición del botón este almacenada en estas propiedades
   3. Añadir los listener necesarios para que cuando cambie el valor de alguna de estas propiedades se actualice el texto del botón (coordenadas (x,y) del Botón en el GridPane)

- El trabajo debe realizarse en el laboratorio.

Ejercicio 2

- El objetivo del proyecto es trabajar con eventos del ratón.
  - Crear un proyecto JavaFX FXML nuevo (es.upv.calculadora)
  - Añadir al grafo de escena los controles y layouts necesario para implementar una calculadora básica (10 dígitos, punto decimal, suma, resta, multiplicación, división, igual y borrado .
  - Añadir los manejadores de eventos necesarios para trabajar con el ratón.

Ejercicio 2- (Posible solución del controlador)

1. Definir las variables de tipo Property adecuadas para almacenar los dos operandos de la calculadora y la posición actual del punto decimal en el operando que estamos introduciendo, o incluso el operador actual .

2. Añadir un manejador para los botones con dígitos sobre el evento MouseCliked (o similar). Cada vez que se ejecute recogerá el digito asociado al botón. En el caso de ser un digito no decimal multiplicaremos el valor anterior del operando por 10 y le sumaremos el digito. En otro caso dividiremos el digito por la posición en la que esta el decimal, sumaremos al operando, y actualizaremos la posición del punto decimal

3. Añadiremos un manejador para el botón del punto decimal. El valor inicial puede ser 0 y cuando se pulse pasará a ser 10.

4. Añadiremos un manejador para el botón de clear. Este reseteara todos los valores a la situación inicial

5. Añadiremos un manejador para los botones de operaciones.

6. Añadiremos un manejador para el botón de igual.

7. Si las operaciones las realizamos sobre variables del tipo property podemos registrar el listener adecuado de tal manera que cada vez que pulsemos un botón se actualice de forma automática la salida por la interfaz.