



## P3. MODELOS Y VISTAS DE DATOS

---

Interfaces Persona Computador

Depto. Sistemas Informáticos y Computación

UPV

# Índice

- Introducción
  - Colecciones en JavaFX
  - Binding de propiedades
  - Componentes de interfaz:
    - ListView
    - ListView con imágenes
  - Paso de parámetros a un controlador
  - Ejercicio
- Componentes gráficos adicionales
    - TableView
    - TableView con imágenes
  - Ejercicio
  - ANEXO I: Aplicaciones con varias ventanas
    - Único stage y varias escenas
    - Varios stages con la correspondiente escena
  - ANEXOII Persistencia

Parte I

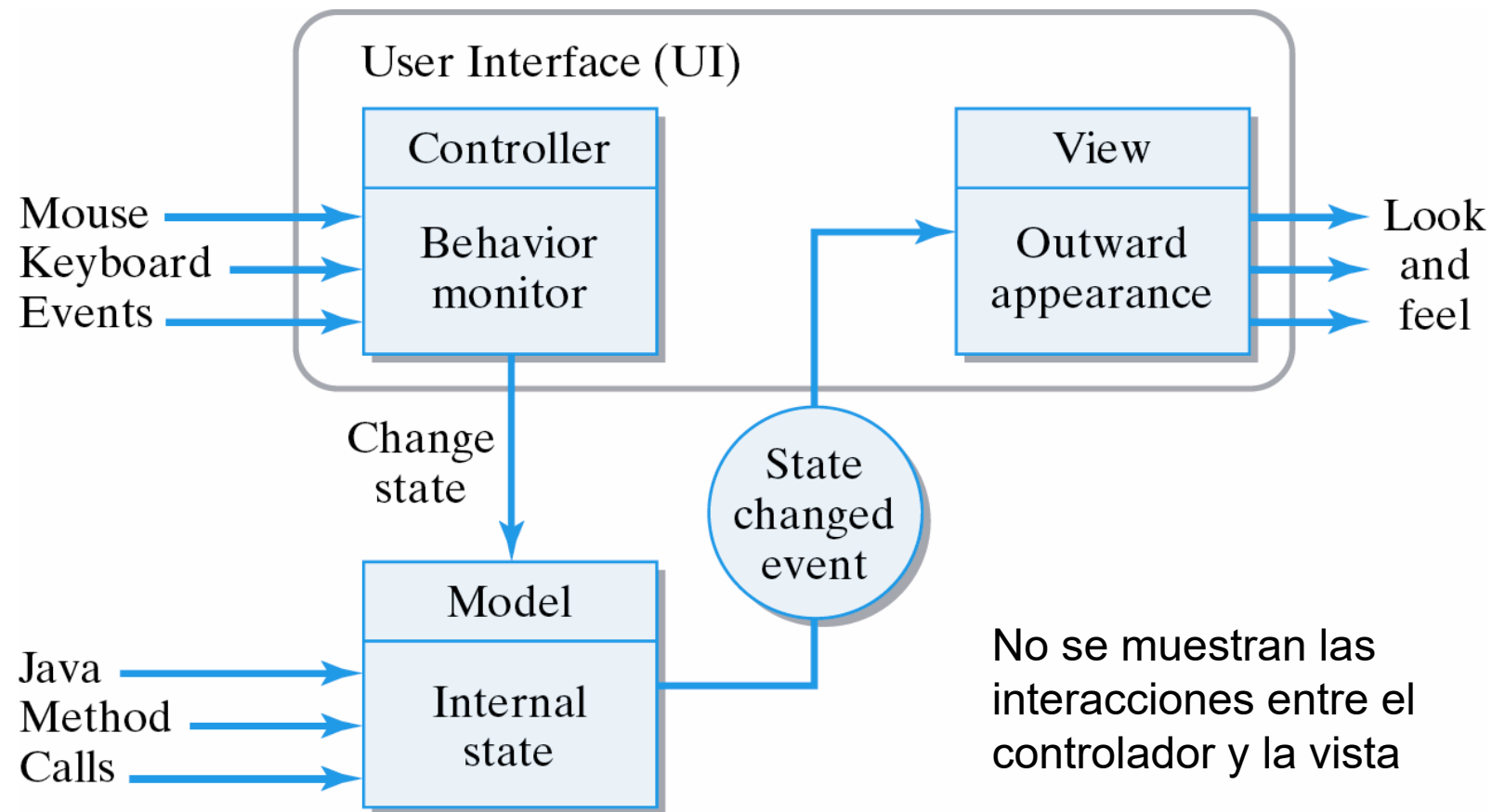
Parte II

# Introducción

- Como se ha mencionado en sesiones previas las aplicaciones modernas pueden estructurarse siguiendo el patrón MVC (Modelo-Vista-Controlador)
- La arquitectura divide al sistema en 3 partes separados:
  - *Vista*: Describe cómo se muestra la información (output/display)
  - *Modelo*: ¿En qué estado está? ¿Qué datos maneja?
  - *Controlador*: ¿Qué entradas del usuario acepta y qué hace con ellas? (entrada/eventos)
- La arquitectura MVC proviene de *Smalltalk-80*, desarrollado durante los años 70.
  - en Smalltalk, MVC se utilizó como un modelo de arquitectura a nivel de aplicación: los datos (*modelo*) se hacen independientes de la UI (*vista* y *controlador*)

# Introducción

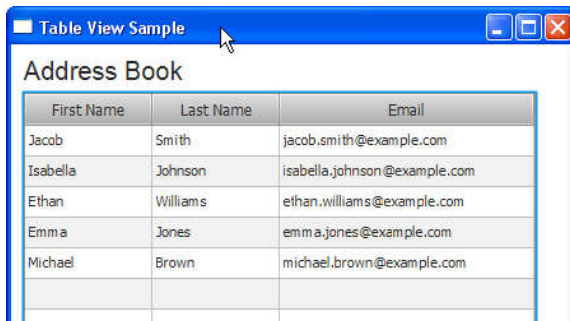
- Relaciones en la arquitectura



# Introducción

- JavaFX contiene controles específicos para presentar datos en una interfaz de usuario:
  - ComboBox<T>, ListView<T>, TableView<T>, TreeTableView<T>
- Podemos separar la definición del componente (**vista**) de los datos (**modelo**) que son visualizados.
- Para el modelo se utilizan **listas observables**

## Vista



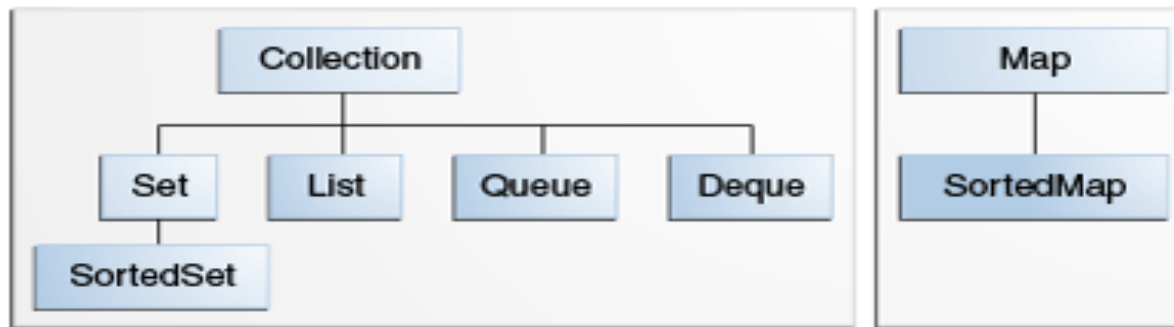
First Name	Last Name	Email
Jacob	Smith	jacob.smith@example.com
Isabella	Johnson	isabella.johnson@example.com
Ethan	Williams	ethan.williams@example.com
Emma	Jones	emma.jones@example.com
Michael	Brown	michael.brown@example.com

## Modelo

```
final ObservableList<Person> data =  
FXCollections.observableArrayList(  
    new Person("Jacob", "Smith", "jacob.smith@example.com"),  
    new Person("Isabella", "Johnson", "isabella.johnson@example.com"),  
    new Person("Ethan", "Williams", "ethan.williams@example.com"),  
    new Person("Emma", "Jones", "emma.jones@example.com"),  
    new Person("Michael", "Brown", "michael.brown@example.com") );
```

# Colecciones en Java

- Las colecciones de **Java** se definen a partir del siguiente conjunto de interfaces:



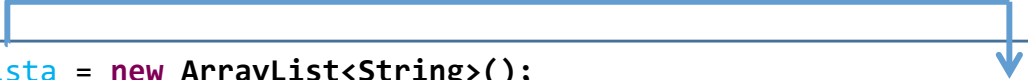
Interface	Hash	Array	Tree	Linked list	Hash+ Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue				LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

# Colecciones en JavaFX

- Además de las colecciones habituales de Java, JavaFX introduce nuevas: `ObservableList`, `ObservableMap`
- Interfaces
  - `ObservableList`: Una lista que permite a los oyentes monitorizar los cambios cuando éstos ocurren.
    - `ListChangeListener`: Una interface que recibe notificaciones de cambios en una `ObservableList`
  - `ObservableMap`: Un mapa que permite a los observadores monitorizar cambios cuando éstos ocurren.
    - `MapChangeListener`: Una interface que recibe notificaciones de cambios en un `ObservableMap`

# Colecciones en JavaFX

- FXCollections: contiene métodos estáticos que permiten envolver colecciones de Java en colecciones JavaFX observables, o crear directamente estas últimas



```
List<String> lista = new ArrayList<String>();
ObservableList<String> listaObservable = FXCollections.observableList(lista);

listaObservable.add("item uno");
lista.add("item dos");
System.out.println("Tamaño FX Collection: " + listaObservable.size());
System.out.println("Tamaño lista: " + lista.size());
```

- La ejecución muestra
- Los elementos que se añaden a la lista son visibles desde la FXCollection

```
Tamaño FX Collection: 2
Tamaño lista: 2
```



# Colecciones en JavaFX

- La colección observable es una clase **envoltorio** de la lista



- Para que los oyentes de la colección JavaFX puedan detectar cambios en la colección los elementos deben añadirse directamente sobre la `listaObservable`

# Colecciones en JavaFX

- Podemos añadir un oyente a la lista observable, permitirá detectar los cambios en la misma

```
listaObservable.addListener(new ListChangeListener<String>() {  
    @Override  
    public void onChanged(ListChangeListener.Change<? extends String> arg0) {  
        System.out.println("Cambio detectado!");  
    }  
});
```

- La ejecución muestra ahora

```
listaObservable.add("item uno");  
lista.add("item dos");  
System.out.println("Tamaño FX Collection: " + listaObservable.size());  
System.out.println("Tamaño lista: " + lista.size());
```



```
<terminated> Main (1) [Java Application]  
Cambio detectado!  
Tamaño FX Collection: 2  
Tamaño lista: 2
```

# Colecciones en JavaFX

- Podemos averiguar el tipo de cambio

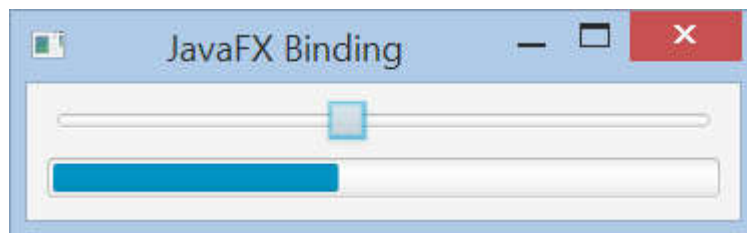
```
listaObservable.addListener(new ListChangeListener<String>() {  
    @Override  
    public void onChanged(ListChangeListener.Change<? extends String> arg0) {  
        System.out.println("Cambio detectado!");  
        while(arg0.next())  
        {  
            System.out.println("Añadido? " + arg0.wasAdded());  
            System.out.println("Eliminado? " + arg0.wasRemoved());  
            System.out.println("Permutado? " + arg0.wasPermutated());  
            System.out.println("Reemplazado? " + arg0.wasReplaced());  
        }  
    }  
});
```

```
listaObservable.add("item uno");  
lista.add("item dos");
```

```
Cambio detectado!  
Añadido? true  
Eliminado? false  
Permutado? false  
Reemplazado? false  
Tamaño FX Collection: 2  
Tamaño lista: 2
```

# Binding de propiedades más ejemplos

- El binding permite sincronizar valores de propiedades, si la propiedad A está **enlazada unidireccionalmente** con la B, cualquier cambio de B se refleja en A. ( $A=f(B)$ )
- Para crear un enlace de **una única vía** usaremos `bind()`, para crearlo **de doble vía** `bindBidirectional()`, para deshacer los enlaces `unbind()` y `unbindBidirectional()`
- Ejemplo: Enlazar la propiedad `progressProperty` de un `ProgressBar` con la propiedad `valueProperty` de un `Slider`



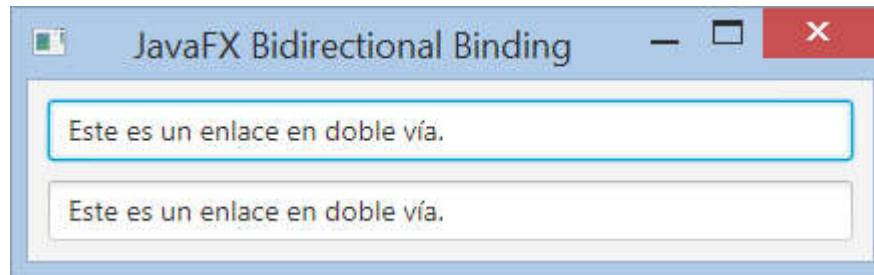
Mientras estén enlazados si se cambia por código el valor de `progressProperty` se produce una excepción

```
@FXML Slider slider; @FXML ProgressBar bar;
```

```
bar.progressProperty().bind(slider.valueProperty());
```

# Binding de propiedades

- Como ejemplo enlazaremos **bidireccionalmente** el contenido de dos campos de texto



```
@FXML TextField tf_1;
```

```
@FXML TextField tf_2;
```

```
// en la inicialización del controlador
```

```
tf_1.textProperty().bindBidirectional(tf_2.textProperty());
```

- Cambios en uno de los campos de texto se transmiten al otro.

# Binding numéricos de propiedades

- Se puede utilizar para enlazar valores de propiedades numéricas

```
IntegerProperty x = new SimpleIntegerProperty(100);  
IntegerProperty y = new SimpleIntegerProperty(200);
```

```
NumberBinding sum = x.add(y);  
int valor = z.intValue();
```

*// sum = x+y genera un error de compilación*

- Para acceder al valor de suma puede utilizarse:  
intValue(), longValue(), floatValue(), doubleValue()  
para obtener los valores como int, long, float y double.
- De manera equivalente

```
IntegerBinding z = (IntegerBinding) x.add(y);  
int valor = z.intValue();
```

# Binding numéricos de propiedades

- Para el ejemplo de círculo en el gridPane, quitamos los oyentes de cambio en anchura y altura y enlazamos la propiedad radio del círculo

```
CirculofxID.radiusProperty().bind(  
    Bindings.min(gridPanefxID.widthProperty(),  
                 gridPanefxID.heightProperty()).divide(5).divide(2));
```

Clase de utilidad

API Fluente, permite  
concatenar operaciones

```
DoubleProperty a = new SimpleDoubleProperty(1.0);  
DoubleProperty b = new SimpleDoubleProperty(2.0);  
DoubleProperty c = new SimpleDoubleProperty(4.0);  
DoubleProperty d = new SimpleDoubleProperty(7.0);  
  
NumberBinding result = Bindings.add (Bindings.multiply(a, b), Bindings.multiply(c,d));  
  
NumberBinding resultado = a.multiply(b).add(c.multiply(d));
```

# Ejemplo ListView

- Las colecciones se emplean para definir el modelo de algunos componentes gráficos.
- ListView

Datos a visualizar

```
ArrayList<String> misdatos = new ArrayList<String>();  
misdatos.add("Java"); misdatos.add("JavaFX");  
misdatos.add("C++");  
misdatos.add("Python"); misdatos.add("JavaScript");  
misdatos.add("C#");
```

Clase envoltorio

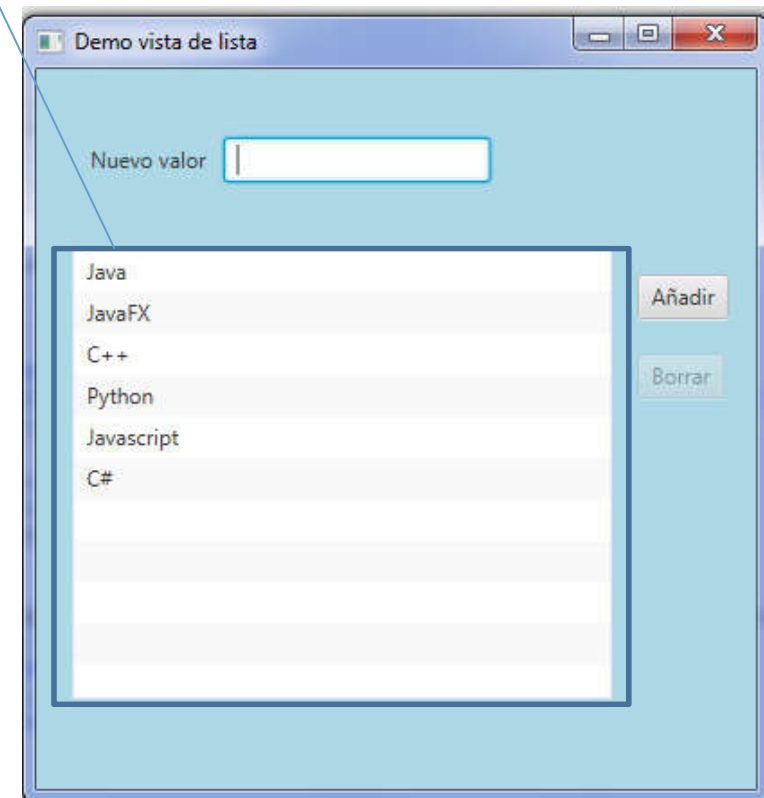
```
private ObservableList<String> datos = null;  
...  
datos = FXCollections.observableArrayList(misdatos);
```

Vinculado a la vista

```
listView.setItems(datos);
```

Cambios en la lista observable automáticamente provocan cambios en la vista: añadir, borrar, etc.

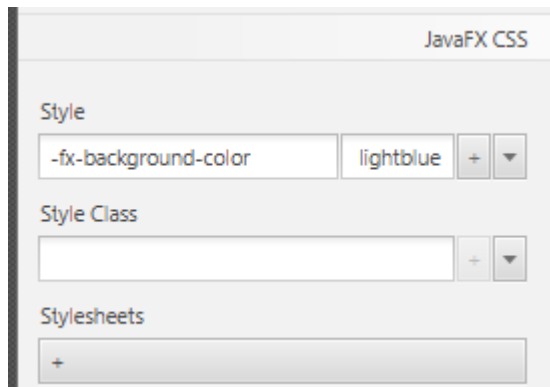
ListView





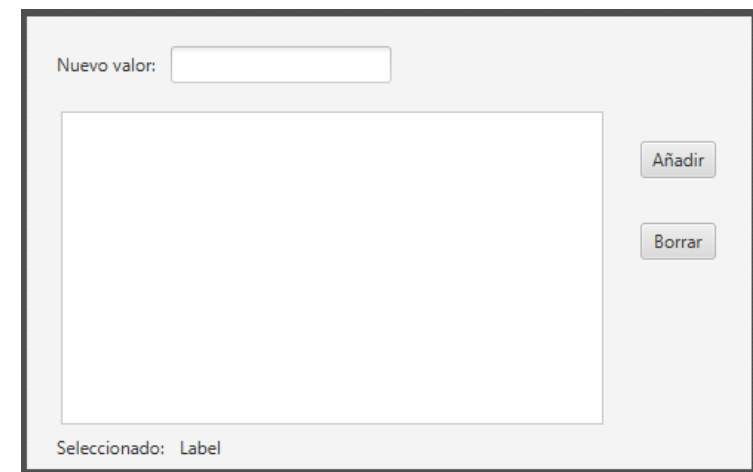
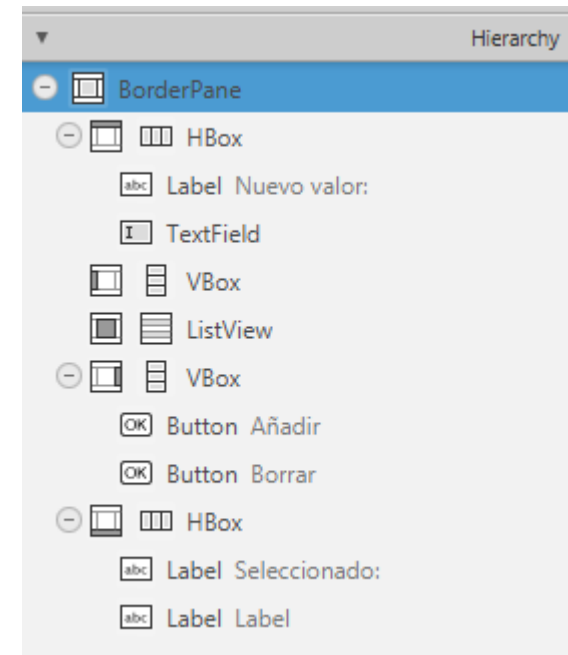
# Ejemplo ListView

- Diseño de la interfaz: BorderPane, con lo controles que se muestran en la captura
- Si queremos poner color a HBox y VBox con hojas de estilo CSS



- Equivalente a poner en el controlador:

```
hBoxfxID.setStyle("-fx-background-color: lightblue;");
```



# Ejemplo ListView

- Métodos útiles en ListView:
  - `getSelectionModel().getSelectedIndex()`: Devuelve el índice del elemento seleccionado de la lista, si ésta está en modo selección simple.
  - `getSelectionModel().getSelectedItem()`: Devuelve el elemento seleccionado.
  - `getFocusModel().getFocusedIndex()`: Devuelve el índice del elemento que tiene el foco.
  - `getFocusModel().getFocusedItem()`: Devuelve el elemento que tiene el foco.
- Para cambiar a modo selección múltiple:

```
getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
```
- Los métodos `getSelectedIndices()` y `getSelectedItems()` de la clase `MultipleSelectionModel` devuelven listas observables que pueden usarse para monitorizar los cambios

<http://docs.oracle.com/javase/8/javafx/user-interface-tutorial/list-view.htm#CEGGEDBF>  
<http://www.java2s.com/Tutorials/Java/JavaFX>

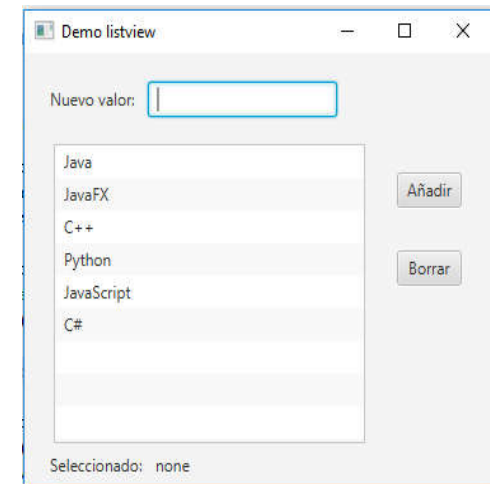
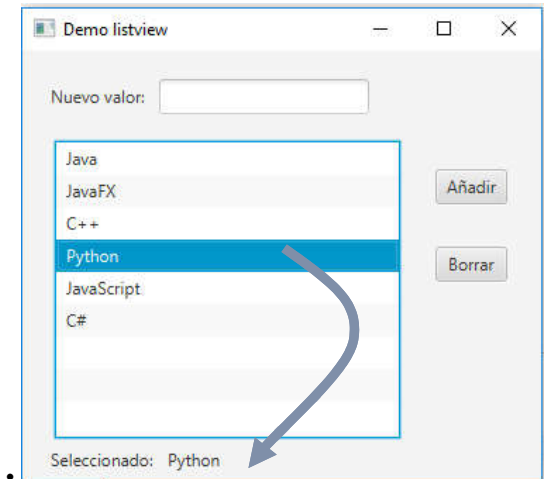
# Ejemplo ListView

- Oyentes de cambios en la selección
- Opción 1: enlace

```
selectedItem.textProperty().bind(  
    listView.getSelectionModel().selectedItemProperty());
```

- Opción 2: oyente de cambio

```
listView.getSelectionModel().selectedIndexProperty().  
    addListener( (o, oldVal, newVal) -> {  
        if (newVal.intValue() == -1)  
            selectedItem.setText("none");  
        else  
            selectedItem.setText(datos.get(newVal.intValue()));  
    });  
selectedItem.setText("none");
```



# Ejemplo ListView

- Opción 3: bind.

```
selectedItem.textProperty().bind(  
    Bindings.when(listView.getSelectionModel().selectedIndexProperty().isEqualTo(-1)).  
    then("ninguno").  
    otherwise(listView.getSelectionModel().selectedItemProperty().asString()));
```

# Ejemplo ListView

- Activación/desactivación de botones al cambiar la selección

```
// The Remove button will be enabled only when an item is selected
botonBorrar.disableProperty().bind(
    Bindings.equal(-1,
        listView.getSelectionModel().selectedIndexProperty()));
```

```
// The Add button will be enabled only then the TextField is not empty
botonAdd.disableProperty().bind(firstNameText.textProperty().isEmpty());
```

- Los botones también puede activarse/desactivarse manualmente mediante:

```
botonAdd.setDisable(true);
botonRemove.setDisable(false);
```

# Ejemplo ListView

- Descargue de Poliformat el ejemplo y póngalo en NetBeans, el proyecto tiene la siguiente estructura:

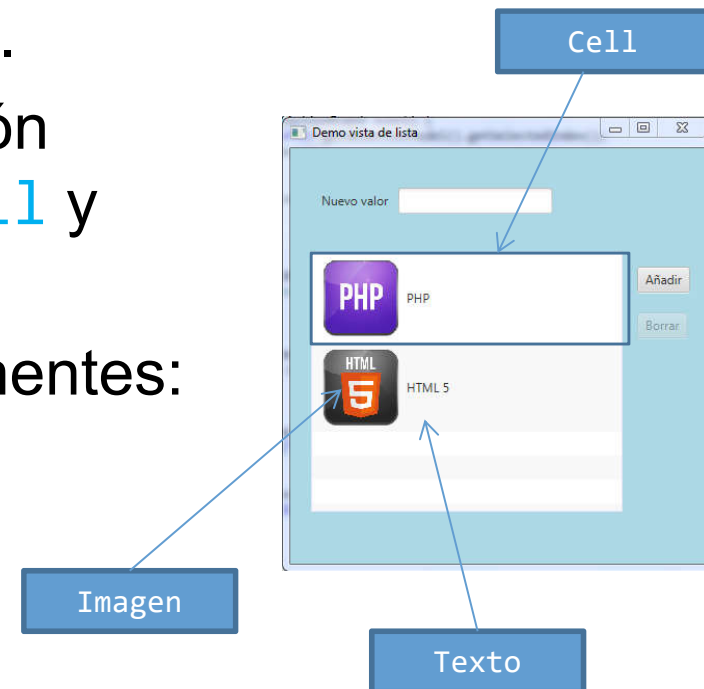
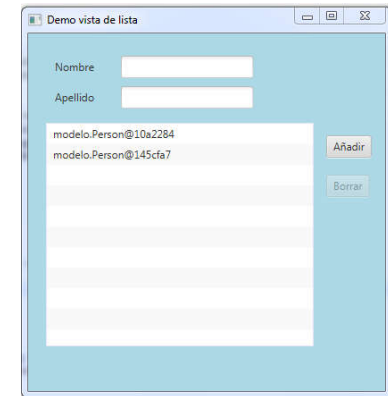


El nombre de los archivos puede aparecer en castellano

- Observe la descomposición en paquetes del proyecto.

# Ejemplo ListView

- Al ejecutar, la captura es la siguiente:
- ListView contiene una visualización por defecto para los Strings, si se recibe un objeto se ejecuta el método `toString`.
- Cuando se necesita una visualización particular se emplean las clases `Cell` y `CellFactory`
- Todo esto es aplicable a los componentes:
  - ComboBox
  - TableView
  - TreeTableView



# ListView: Cell y CellFactory

- Para la clase *Persona* tengo que indicar qué quiero que se muestre en el listView.

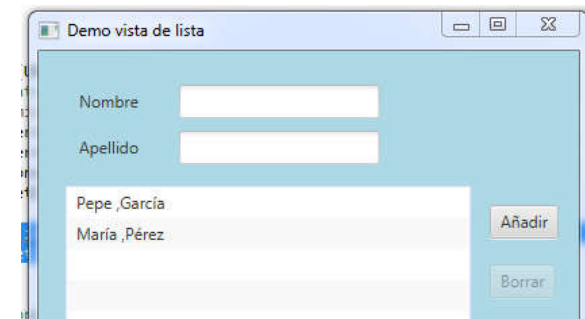
La clase para las celdas del listView

```
// Clase local al controlador
class PersonListCell extends ListCell<Person>
{
    @Override
    protected void updateItem(Person item, boolean empty)
    { super.updateItem(item, empty); // Obligatoria esta llamada
      if (item==null || empty) setText(null);
      else setText(item.getFirstName() + " , " + item.getLastName());
    }
}
```

La información en pantalla

- En el initialize del controlador fijo la factoría de celdas

```
// en el código de inicialización del controlador
listView.setCellFactory(c-> new PersonListCell());
```

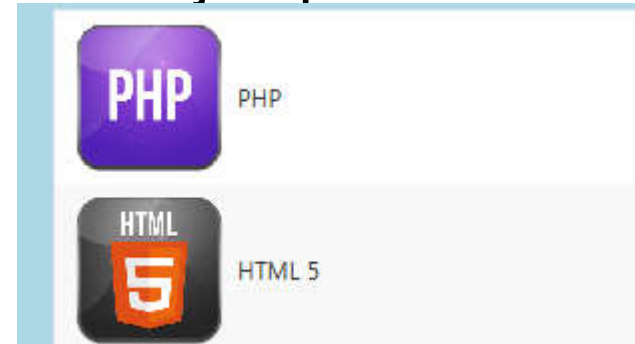




# ListView: Cell y CellFactory

- Si queremos añadir una imagen, en el ejemplo inicial de los lenguajes.

```
// Clase local al controlador
class LenguajeListCell extends ListCell<Lenguaje>
{
    private ImageView view = new ImageView();
    @Override
    protected void updateItem(Lenguaje item, boolean empty)
    {
        super.updateItem(item, empty);
        if (item==null || empty) {
            setText(null);
            setGraphic(null);}
        else {
            view.setImage(item.getImagen());
            setGraphic(view);
            setText(item.getNombre());
        }
    }
}
```



```
package modelo;
```

```
public class Lenguaje {
    private String nombre;
    private Image imagen;
```

```
...
```

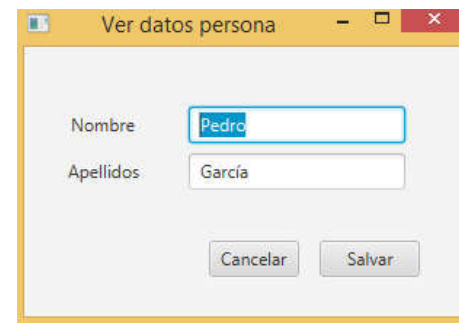
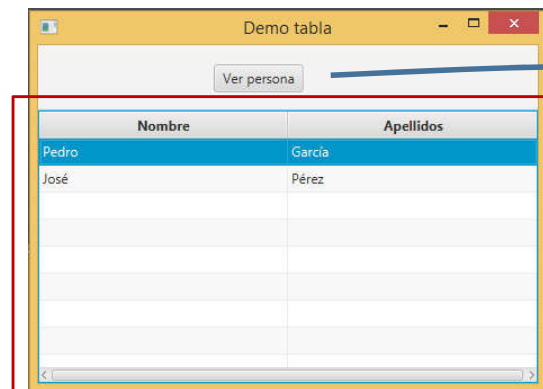
- En el initialize del controlador:

```
listView.setCellFactory(c-> new LenguajeListCell());
```

# Paso de datos a un controlador

- Supongamos que necesitamos un formulario para mostrar información de una persona, pasando como parámetro el nombre y los apellidos

TableView explicado más adelante



```
public class DatosPersonaControlador {  
  
    @FXML private TextField nombrefxID;  
    @FXML private TextField apellidosfxID;  
  
    public void initPersona( Person p)  
    { nombrefxID.setText(p.getFirstName());  
      apellidosfxID.setText(p.getLastName());  
    }  
}
```

Se llama desde el controlador de la ventana principal

# Paso de datos a un controlador

- Al cargar el fxml del formulario podemos acceder a su controlador e invocar el método que hemos llamado `initPersona`

```
//AnchorPane root = (AnchorPane)FXMLLoader.load(getClass().getResource("/vista/DatosPersona.fxml"));
```



Cambiar por acceso no estático

```
FXMLLoader miCargador = new FXMLLoader(getClass().getResource("/vista/DatosPersona.fxml"));  
AnchorPane root = (AnchorPane) miCargador.load();
```

```
// acceso al controlador de datos persona  
DatosPersonaControlador controladorPersona = miCargador.<DatosPersonaControlador>getController();  
controladorPersona.initPersona(persona);
```

```
Scene scene = new Scene(root,400,400);  
Stage stage = new Stage();  
stage.setScene(scene);  
stage.setTitle("Ver datos persona");  
stage.initModality(Modality.APPLICATION_MODAL);  
stage.showAndWait();
```

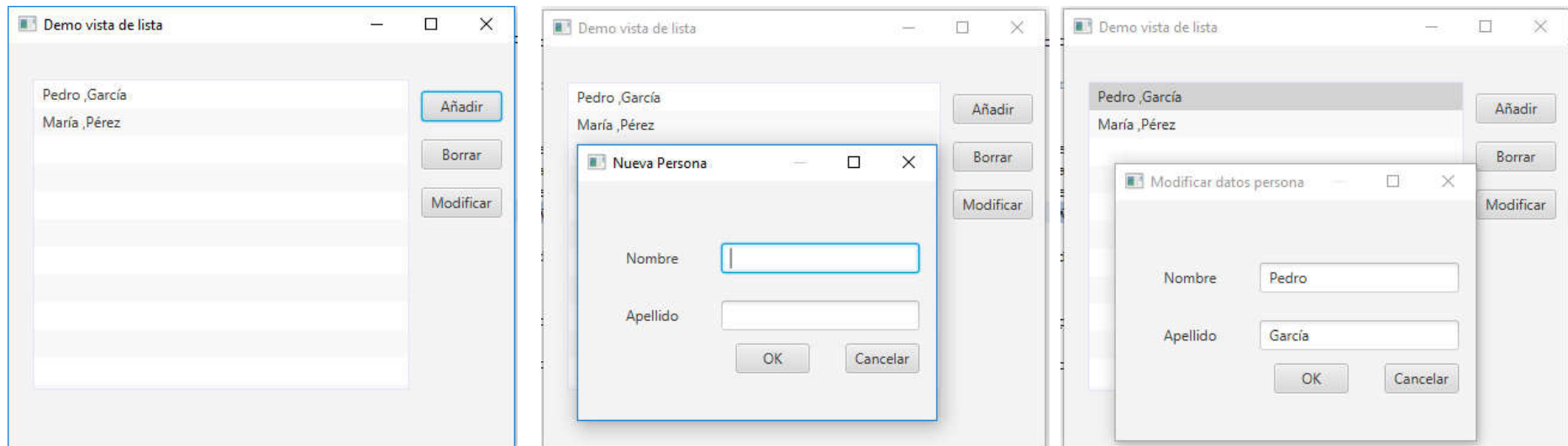
Indica que la nueva ventana es modal

Muestra la nueva ventana y espera a que se cierre

- El código anterior está en el manejador del botón Ver Persona.

# Ejercicio

- Para realizar en el laboratorio
- A partir del proyecto de la ListView con la clase Persona:
  - Crear una nueva interfaz como la de la figura.
  - Hacer que el botón Añadir esté siempre habilitado.
  - Añadir un botón Modificar (habilitado si algo está seleccionado)
  - Al pulsar el botón Modificar o Añadir debe mostrarse otra ventana para que en un caso se modifiquen los datos y en el otro se añadan.

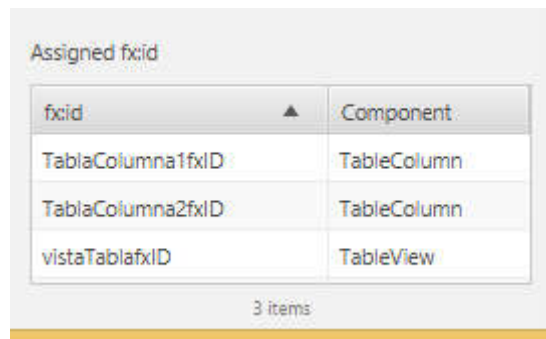
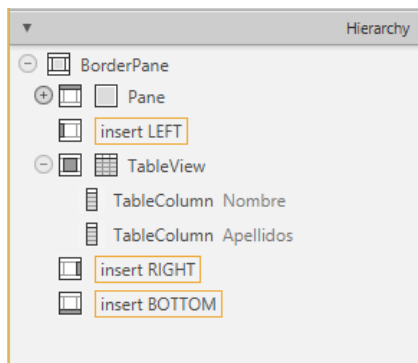


# PARTE 2

---

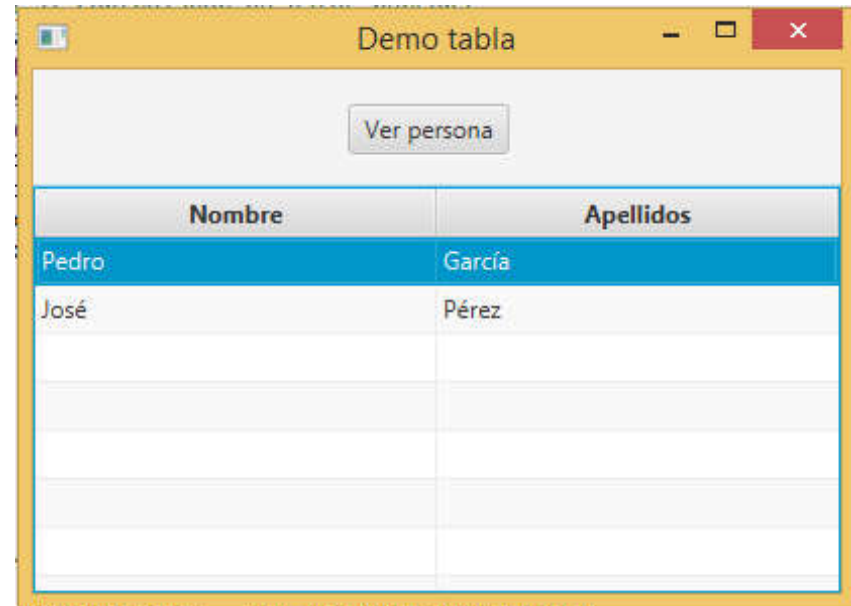
# TableView

- El control está diseñado para visualizar filas de datos divididos en columnas
- **TableColumn** representa una columna de la tabla y contiene **CellValueFactory** para visualizaciones especiales, como imágenes
- Para una tabla que contenga texto en las columnas
  - Scene Builder



# TableView

- La tabla contiene instancias de la clase Persona.
- Las columnas son el nombre y los apellidos



```
public class Person {  
  
    private StringProperty firstName = new SimpleStringProperty();  
    private StringProperty lastName = new SimpleStringProperty();  
  
    public Person(String firstName, String lastName) {  
        this.firstName.setValue(firstName);  
        this.lastName.setValue(lastName);  
    }  
}
```

# TableView

- Para indicar cómo se pueblan las celdas de una columna se usa el método: `setCellValueFactory` de `TableColumn`

- Código en el controlador

```
private ObservableList<Person> myData;
```

```
@FXML private TableView<Person> tableView;
```

```
@FXML private TableColumn<Person, String> firstNameColumn;
```

```
@FXML private TableColumn<Person, String> lastNameColumn;
```

- Código de inicialización en el controlador

```
firstNameColumn.setCellValueFactory(  
    new PropertyValueFactory<Person, String>("firstName"));
```

```
lastNameColumn.setCellValueFactory(  
    new PropertyValueFactory<Person, String>("lastName"));
```

```
tableView.setItems(myData);
```



# TableView

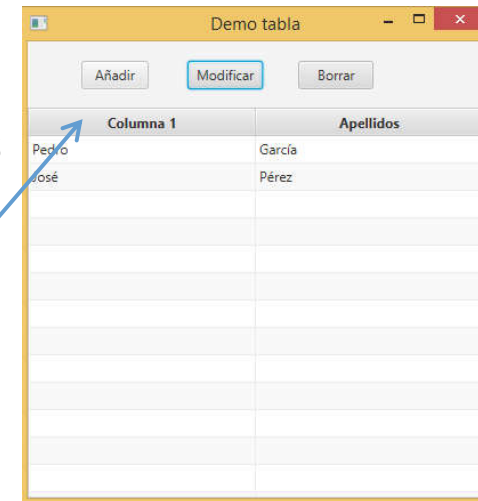
- La tabla contiene instancias de la clase Persona.
- Las columnas son el nombre y los apellidos

```
public class Person {  
  
    private StringProperty firstName = new SimpleStringProperty();  
    private StringProperty lastName = new SimpleStringProperty();  
  
    public Person(String firstName, String lastName) {  
        this.firstName.setValue(firstName);  
        this.lastName.setValue(lastName);  
    }  
}
```

## Código en el controlador

```
firstNameColumn.textProperty().set("Columna 1");
firstNameColumn.setCellValueFactory(
    new PropertyValueFactory<Person, String>("firstName"));
```

Indica el valor que irá en la columna



# TableView

- El código:

```
firstNameColumn.setCellValueFactory(  
    new PropertyValueFactory<Person, String>("firstName"));
```

- Equivale a:

```
firstNameColumn.setCellValueFactory(new Callback<CellDataFeatures<Person, String>,  
    ObservableValue<String>>()  
{  
    public ObservableValue<String> call(CellDataFeatures<Person, String> p) {  
        return p.getValue().firstNameProperty();  
    }  
});
```

- `CellValueFactory` indica el valor que irá en la columna,  
`CellFactory` indica cómo se presentará en pantalla.

# TableView con imágenes

- Modificamos la tabla para que muestre una imagen y un campo (ciudad) que está en otra clase.

```
public class Person {  
    private final StringProperty fullName = new SimpleStringProperty();  
    private final IntegerProperty id = new SimpleIntegerProperty();  
    private final ObjectProperty<Residence> residence = new SimpleObjectProperty<>();  
    private final StringProperty pathImage = new SimpleStringProperty();  
}
```

# TableView con imágenes

- Campos inyectados

```
@FXML private TableColumn<Person, Integer> idColumn;  
@FXML private TableColumn<Person, String> fullNameColumn;  
@FXML private TableColumn<Person, Residence> cityColumn;  
@FXML private TableColumn<Person, String> imageColumn;  
@FXML private TableView<Person> tableView;
```

- En el controlador, en la inicialización

```
idColumn.setText("DNI");  
fullNameColumn.setText("Nombre y Apellidos");  
cityColumn.setText("Ciudad");  
imageColumn.setText("Imagen");  
// Para que se vean las columnas.  
idColumn.setCellValueFactory(  
    new PropertyValueFactory<Person, Integer>("id"));  
fullNameColumn.setCellValueFactory(  
    new PropertyValueFactory<Person, String>("fullName"));
```

# TableView con imágenes

- Para la ciudad que es un campo de Residencia, también en la inicialización del controlador

```
// ¿Qué información se visualiza?  
cityColumn.setCellValueFactory(cellData3 -> cellData3.getValue().residenceProperty());  
  
// ¿Cómo se visualiza la información?  
// si quiero únicamente un string no pongo el setCellFactory  
cityColumn.setCellFactory(v -> {  
    return new TableCell<Person, Residence>() {  
        @Override  
        protected void updateItem(Residence item, boolean empty) {  
            super.updateItem(item, empty);  
            if (item == null || empty) setText(null);  
            else setText("-->" + item.getCity());  
        }  
    };  
});
```

Getter de las clases anteriores

Debe ser siempre un valor observable

Visualización elegida

Declarado igual que la columna correspondiente

```
@FXML private TableColumn<Person, Residence> cityColumn;
```

# TableView con imágenes

- Para la columna que contiene la imagen

```
imageColumn.setCellValueFactory(celda4 -> celda4.getValue().pathImageProperty());
imageColumn.setCellFactory(columna -> {
    return new TableCell<Person,String> () {
        private ImageView view = new ImageView();
        @Override
        protected void updateItem(String item, boolean empty) {
            super.updateItem(item, empty);
            if (item == null || empty) setGraphic(null);
            else {
                Image image = new Image(MainWindowController.class.getResourceAsStream(item),
                                         40, 40, true, true);
                view.setImage(image);
                setGraphic(view);
            }
        }
    };
}); //setCellFactory
```

Carga el archivo png de la imagen.  
item contiene el path

- El código anterior funciona si la imagen se encuentra en la carpeta resources del proyecto, en otro caso usar el código de la siguiente transparencia

# TableView con imágenes

- Si la imagen se encuentra en una ubicación del disco duro fuera del jar del proyecto

```
imageColumn.setCellFactory(columna -> {  
    return new TableCell<Person,String> () {  
        private ImageView view = new ImageView();  
        @Override  
        protected void updateItem(String item, boolean empty) {  
            super.updateItem(item, empty);  
            if (item == null || empty) setGraphic(null);  
            else {  
                File imageFile = new File(item);  
                //item path y nombre del archivo  
                String fileLocation = imageFile.toURI().toString();  
                Image image = new Image(fileLocation,40,40,true,true);  
                view.setImage(image);  
                setGraphic(view);  
            }  
        }  
    };  
});
```

# TableView con atributos

- Supongamos que la definición de la clase Person contiene una propiedad y 3 atributos

```
public class Person {  
    private StringProperty fullName = new SimpleStringProperty();  
    private int id; // atributo, no propiedad  
    private Residencia residence; // no propiedad  
    private String pathImage; // no propiedad  
    ..}
```

```
public class Residencia {  
    private final String city;  
    private final String province;  
    ..}
```

- Los campos inyectados ahora son:

```
@FXML private TableColumn<Person, Integer> idColumn;  
@FXML private TableColumn<Person, String> fullNameColumn;  
@FXML private TableColumn<Person, String> cityColumn;  
@FXML private TableColumn<Person, String> imageColumn;  
@FXML private TableView<Person> tableView;
```



# TableView con atributos

- Para visualizar la propiedad y los 3 atributos

```
idColumn.setCellValueFactory(cellData -> new  
    SimpleIntegerProperty(cellData.getValue().getId()).asObject());
```

```
fullNameColumn.setCellValueFactory(cellData ->  
    cellData.getValue().fullNameProperty());
```

```
cityColumn.setCellValueFactory( cellData -> new  
    SimpleStringProperty(cellData.getValue().getResidence().getCity()));
```

```
imageColumn.setCellValueFactory(cellData -> new  
    SimpleStringProperty(cellData.getValue().getPathImagen()));
```

- Para propiedades la expresión siguiente no genera ni errores de compilación, ni de ejecución, en el caso de que el nombre de la propiedad no exista. El efecto es que no muestra nada en la columna. Utilizar en su lugar la enmarcada de arriba.

```
fullNameColumn.setCellValueFactory(  
    new PropertyValueFactory<Person, String>("fullName"));
```

# Ejercicio

- A partir del proyecto de la ListView con la clase Persona, cambie la interfaz para que muestre la lista de personas en un TableView.
- Inicialice la lista de personas en el main y pase los datos al controlador.
- Añada a la interfaz los botones: Añadir, Borrar y Modificar.
  - En el caso de modificar y añadir debe mostrarse una ventana emergente para que en un caso se modifiquen los datos y en el otro se añadan.
- A realizar en el laboratorio al final de la sesión

# Ejercicio continuación...

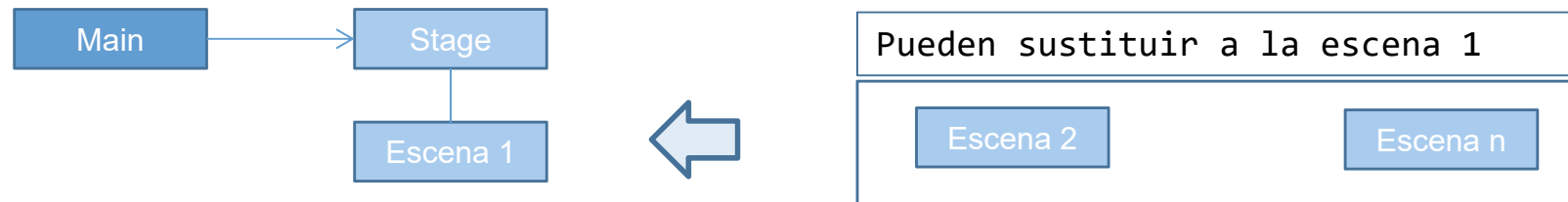
- Si terminó el ejercicio, modifíquelo para que la tabla muestre una imagen junto a cada persona.
- Las 3 imágenes están en un archivo zip de poliformat.
- En el proyecto NetBeans incluya un paquete con los 3 archivos png en un paquete recursos. Los path de las imágenes se indican:

`"/recursos/Sonriente.png"`

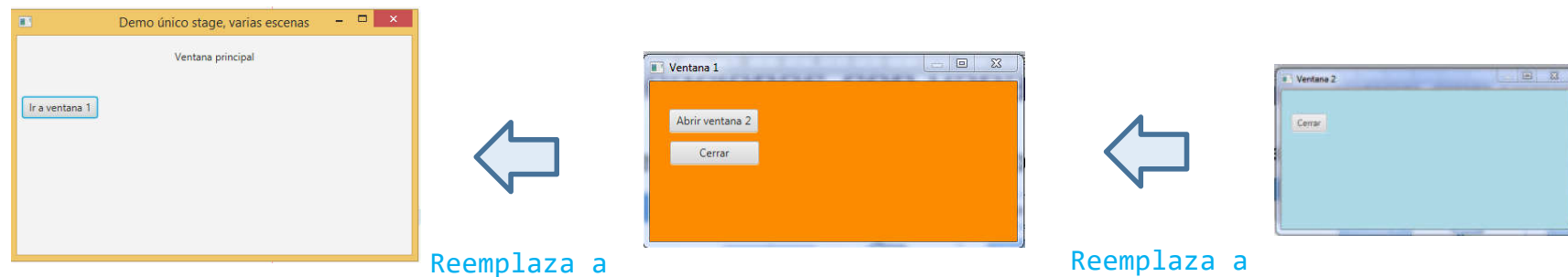
```
new Person("Juan Gómez", 45678912,  
    new Residence("Valencia", "Valencia"), "/recursos/Sonriente.png")
```

# Anexo I: aplicaciones con varias ventanas

- Podemos tener un **único Stage** con varias escenas



- La aplicación tiene visible una única ventana (Stage)



- A cada ventana se pasa el Stage y la escena, cada controlador carga la siguiente escena

# Varias ventanas: único stage

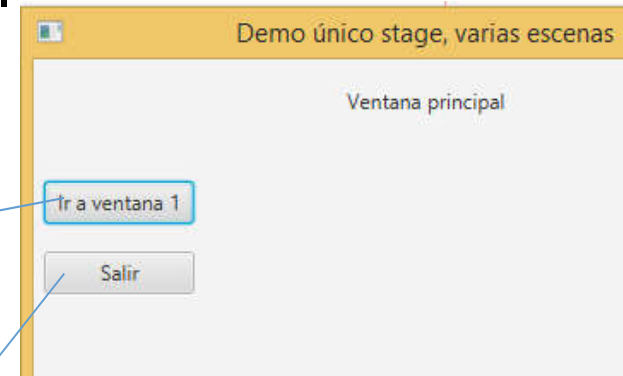
- El stage se pasa como parámetro a los controladores de cada una de las ventanas

```
public class StageUnico extends Application {  
  
    @Override  
    public void start(Stage stage) throws Exception {  
  
        FXMLLoader loader = new FXMLLoader(getClass().getResource("/vista/Principal.fxml"));  
        BorderPane root = (BorderPane) loader.load();  
        Scene scene = new Scene(root);  
        stage.setScene(scene);  
        stage.setTitle("Demo único stage, varias escenas");  
        PrincipalControlador controladorPrincipal = loader.<PrincipalControlador>getController();  
        controladorPrincipal.initStage(stage);  
        |  
        stage.show();  
    }  
  
    /**  
    * @param args the command line arguments  
    */  
    public static void main(String[] args) {  
        launch(args);  
    }  
}
```

# Varias ventanas: único stage

- Código del controlador ventana principal

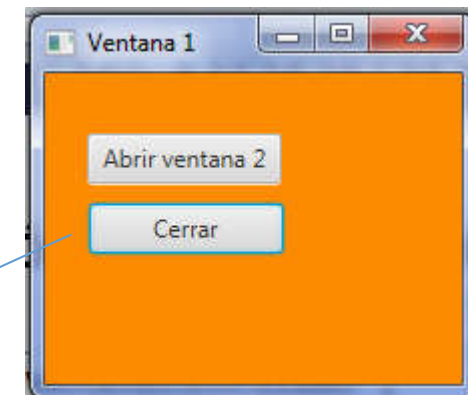
```
public class PrincipalControlador {  
    private Stage primaryStage;  
  
    public void initStage( Stage stage)  
    { primaryStage = stage;}  
  
    @FXML private void irAVentana1(ActionEvent event) {  
        try {  
            FXMLLoader miCargador = new  
                FXMLLoader(getClass().getResource("/vista/Ventana1.fxml"));  
            Parent root = (Parent) miCargador.load();  
            // acceso al controlador de ventana 1  
            Ventana1Controlador ventana1 = miCargador.<Ventana1Controlador>getController();  
            ventana1.initStage(primaryStage);  
            Scene scene = new Scene(root);  
            primaryStage.setScene(scene);  
            primaryStage.show();  
        } catch (IOException e) {e.printStackTrace();}  
    }  
  
    @FXML private void salir(ActionEvent event) {  
        primaryStage.hide();  
    }  
}
```



# Varias ventanas: único stage

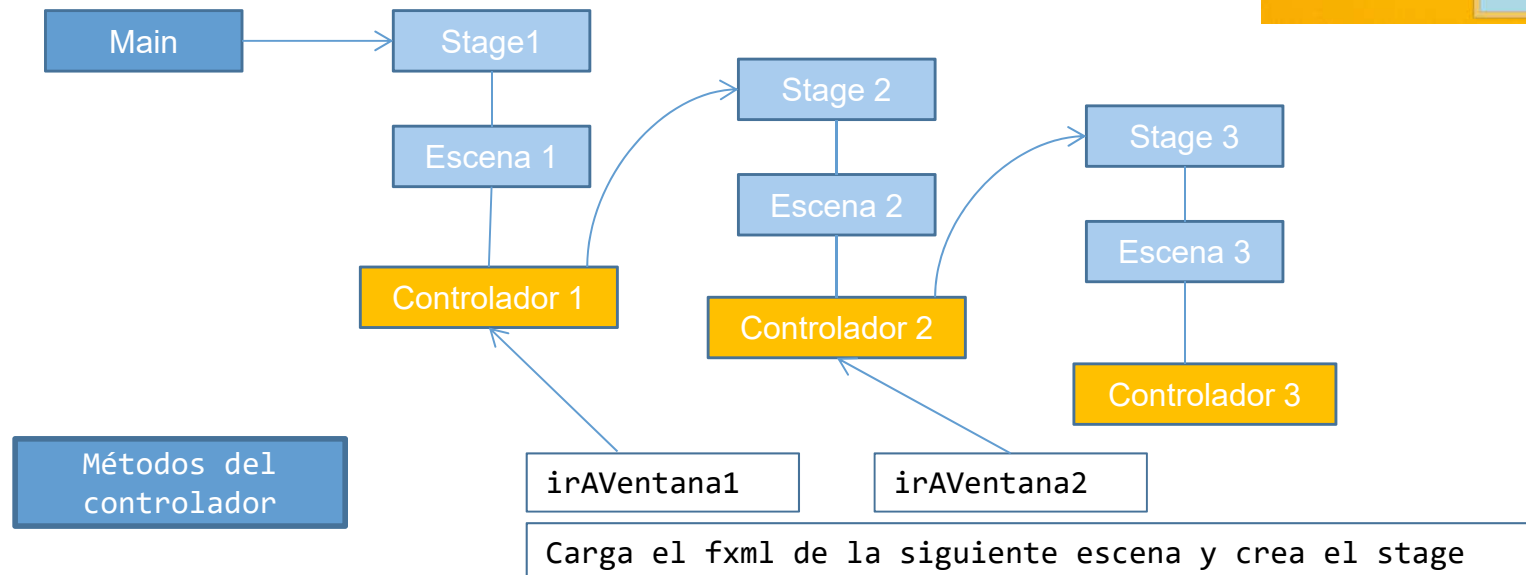
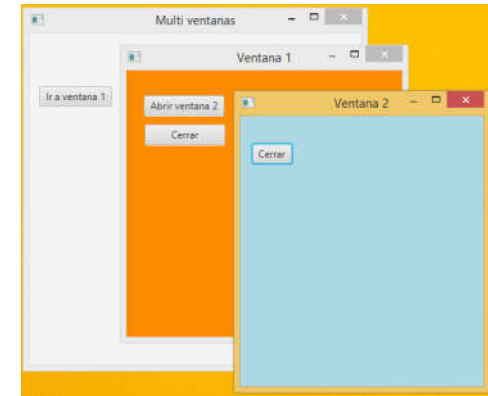
- El código del controlador de la ventana 1:

```
public class Ventana1Controlador {  
    private Stage primaryStage;  
    private Scene escenaAnterior;  
    private String tituloAnterior;  
  
    public void initStage(Stage stage)  
    { primaryStage = stage;  
      escenaAnterior = stage.getScene();  
      tituloAnterior = stage.getTitle();  
      primaryStage.setTitle("Window 1");  
    }  
  
    // similar a ir aVentana1  
    @FXML private void irAVentana2(ActionEvent event) {}  
  
    @FXML private void cerrarAccion(ActionEvent event) {  
        System.out.println("Cerrando ventana 1");  
        primaryStage.setTitle(tituloAnterior);  
        primaryStage.setScene(escenaAnterior);  
    }  
}
```



# Aplicaciones con varias ventanas

- Podemos usar **varios stages** y cada uno con una escena
- Las tres ventanas están visibles
- Se definen modales, salvo la inicial
- Cada controlador carga el siguiente Stage





# Aplicaciones con varias ventanas

- El código del main es similar al ejemplo anterior.
- Cada ventana (escena) tiene su Stage

```
public class Main extends Application {  
    @Override  
    public void start(Stage primaryStage) {  
        try {  
            FXMLLoader miCargador = new FXMLLoader(getClass().getResource("/vista/Principal.fxml"));  
            BorderPane root = (BorderPane) miCargador.load();  
            Scene scene = new Scene(root,400,400);  
            primaryStage.setTitle("Multi ventanas");  
            primaryStage.setScene(scene);  
            primaryStage.show();  
        } catch(Exception e) {  
            e.printStackTrace();  
        }  
    }  
  
    public static void main(String[] args) {  
        launch(args);  
    }  
}
```

# Aplicaciones con varias ventanas

- Controlador principal

```
public class PrincipalControlador implements Initializable {
```

```
    @FXML private void irAVentana1(ActionEvent event) {
```

```
        try {
```

```
            Stage estageActual = new Stage();
```

Ventana 1

```
            FXMLLoader miCargador = new FXMLLoader(getClass().getResource("/vista/Ventana1.fxml"));
```

```
            Parent root = (Parent) miCargador.load();
```

```
            miCargador.<Ventana1Controlador>getController().initStage(estageActual);
```

```
            Scene scene = new Scene(root,400,400);
```

```
            estageActual.setScene(scene);
```

```
            estageActual.initModality(Modality.APPLICATION_MODAL);
```

Modalidad

```
            estageActual.show();
```

```
        } catch (IOException e) {
```

```
            e.printStackTrace();
```

```
        }
```

```
    }
```

```
    @FXML void salirAccion(ActionEvent event) {
```

```
        Node n = (Node)event.getSource();
```

```
        n.getScene().getWindow().hide();
```


```
    }
```

```
}
```

# Aplicaciones con varias ventanas

- Código controlador ventana 1

```
public class Ventana1Controlador implements Initializable {
    private Stage primaryStage;
    public void initStage(Stage stage) {
        primaryStage = stage;
        primaryStage.setTitle("Ventana 1");
    }
    @FXML private void irAVentana2(ActionEvent event) {
        try { Stage estageActual = new Stage();
            FXMLLoader miCargador = new FXMLLoader(getClass().getResource("/vista/Ventana2.fxml"));
            Parent root = (Parent) miCargador.load();
            miCargador.<Ventana2Controlador>getController().initStage(estageActual);
            Scene scene = new Scene(root,400,400);
            estageActual.setScene(scene);
            estageActual.initModality(Modality.APPLICATION_MODAL);
            estageActual.show();
        } catch (IOException e) {e.printStackTrace();}
    }
    @FXML private void cerrarAccion(ActionEvent event) {
        Node minodo = (Node) event.getSource();
        minodo.getScene().getWindow().hide();
        System.out.println("Cerrando ventana 1");
    }
}
```



## ANEXO II. Persistencia (I)

¿Por qué usar XML y no bases de datos?

- Son una de las formas más habituales de almacenar información.
- Habitualmente organizan los datos según el modelo relacional (tablas relacionadas mediante índices).
- Para los casos de estudio, dada su sencillez, usaremos XML para almacenar la información.

# Persistencia (II)

¿Por qué usar XML?

- Es más fácil para nuestro sencillo modelo de datos.
- Librería JAXB (*Java Architecture for XML Binding*).
- Con pocas líneas de código JAXB se puede generar esta salida en XML:

```
<persons>
  <person>
    <birthday>1999-02-21</birthday>
    <city>some city</city>
    <firstName>Hans</firstName>
    <lastName>Muster</lastName>
    <postalCode>1234</postalCode>
    <street>some street</street>
  </person>
</persons>
```

```
<person>
  <birthday>1999-02-21</birthday>
  <city>some city</city>
  <firstName>Anna</firstName>
  <lastName>Best</lastName>
  <postalCode>1234</postalCode>
  <street>some street</street>
</person>
</persons>
```

# Persistencia (III)

## JAXB

- Incluido en el JDK. No se necesita añadir ninguna librería adicional.
- Proporciona dos funcionalidades principales:
  - **Marshalling**: conversión de objetos Java a XML.
  - **Unmarshalling**: conversión de XML a objetos Java.



- Lo único que se necesita es añadir anotaciones al texto de las clases Java.

# Persistencia (IV)

- Las anotaciones necesarias son las siguientes:

Anotación	Significado
<code>@XmlAccessorType(PUBLIC_MEMBER, PROPERTY, FIELD, o NONE)</code>	Tipo de acceso para enlazar con el XML a las propiedades y campos de la clase
<code>@XmlRootElement(namespace = "namespace")</code>	Define la raíz del archivo XML
<code>@XmlType(propOrder = { "field2", "field1",.. })</code>	Indica el orden en el que se salvarán los atributos de la clase
<code>@XmlElement (name = "nombre")</code>	Indica el atributo que se salvará
<code>@XmlAttribute (name = "nombre")</code>	Especifica un atributo para el elemento raíz del XML
<code>@XmlTransient</code>	Atributo que no se salvará

# Ejemplos en JAXB

- Supongamos que tenemos la clase

```
public class Person {  
  
    private final StringProperty fullName = new SimpleStringProperty();  
    private final IntegerProperty id = new SimpleIntegerProperty();  
    private final List<Residence> residences;  
    private final StringProperty pathImage = new SimpleStringProperty();  
}
```

- Y queremos persistir en XML una de sus instancias.
- Procedimiento:
  - Añadir un constructor sin parámetros a **Person**
  - Etiquetar la clase con **@XmlElement**
  - **Residence**, aunque es otra clase a salvar, no necesita anotaciones ya que no será la raíz de un archivo XML.



# Ejemplos en JAXB

- Para la clase:

Única anotación necesaria

@XmlElement

public class Person {

Información a salvar en el XML

private final StringProperty fullName = new SimpleStringProperty();

private final IntegerProperty id = new SimpleIntegerProperty();

private final List<Residence> residences;

private final StringProperty pathImage = new SimpleStringProperty();

public Person() { }

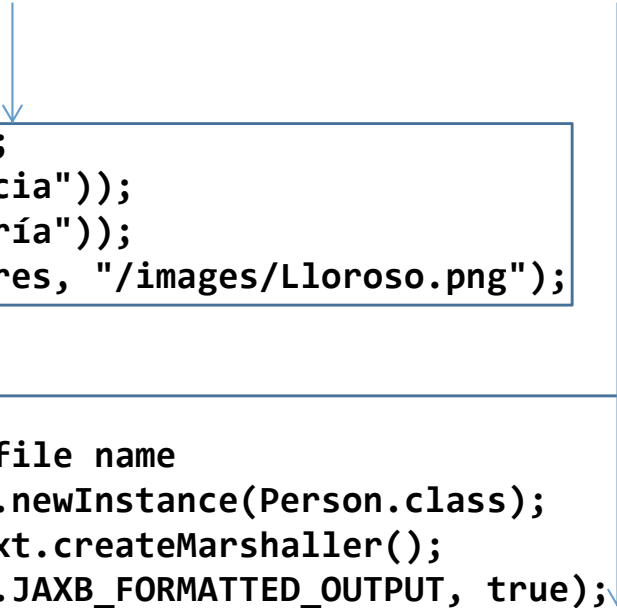
...// Todos los atributos y propiedades tienen getters y setters

- Cuando no se indica @XmlAccessorType el acceso es a todas las propiedades y atributos públicos (los que tienen getters y setters aunque se hayan declarado como atributos privados)
- Si quiere ponerse un nombre al elemento raíz distinto del nombre de la clase, se usa la anotación:

@XmlElement ( name = "person" )

# Ejemplos en JAXB

- Una vez creada una instancia de Persona, ¿Cómo se salva?



```
List<Residence> res = new ArrayList<>();
res.add(new Residence("Museros", "Valencia"));
res.add(new Residence("Roquetas", "Almería"));
Person p = new Person(100, "John Doe", res, "/images/Lloroso.png");
```

// Salvar en disco.

```
try {
    File file = new File("ddb.xml"); // file name
    JAXBContext jaxbContext = JAXBContext.newInstance(Person.class);
    Marshaller jaxbMarshaller = jaxbContext.createMarshaller();
    jaxbMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
    jaxbMarshaller.marshal(p, file); // save to a file
    jaxbMarshaller.marshal(p, System.out); // echo to the console
} catch (JAXBException e) {
    e.printStackTrace();
}
```

# Ejemplos en JAXB

- Contenido del archivo en disco:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<person>
  <fullName>John Doe</fullName>
  <id>100</id>
  <pathImage>/images/Lloroso.png</pathImage>
  <residences>
    <city>Museros</city>
    <province>Valencia</province>
  </residences>
  <residences>
    <city>Roquetas</city>
    <province>Almería</province>
  </residences>
</person>
```

# Ejemplos en JAXB

- El archivo XML puede ser leído por el programa para crear una instancia de persona.

```
try {  
    File file = new File("ddbb.xml");  
    JAXBContext jaxbContext = JAXBContext.newInstance(Person.class);  
    Unmarshaller jaxbUnmarshaller = jaxbContext.createUnmarshaller();  
    Person person = (Person) jaxbUnmarshaller.unmarshal(file);  
} catch (JAXBException e) {  
    e.printStackTrace();  
}
```

# Ejemplos en JAXB

- Ahora queremos salvar en XML una lista de personas.
- Las FXCollections de Java no se pueden mapear a XML directamente.
- Creamos una clase que contenga la lista de personas.

```
@XmlRootElement
public class ListPersonWrapper {
    private List<Person> personList;
    public ListPersonWrapper() { }
    @XmlElement(name = "Person")
    public List<Person> getPersonList() {
        return personList;
    }
    public void setPersonList(List<Person> list) {
        personList = list;
    }
}
```

# Ejemplos en JAXB

- El código para salvar a XML es similar al ejemplo anterior

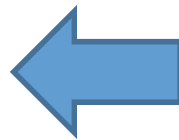
```
ListPersonWrapper listToSave = new ListPersonWrapper();
listToSave.setPersonList(theList);

try {
    File file = new File("persons.xml");
    JAXBContext jaxbContext =
JAXBContext.newInstance(ListPersonWrapper.class);
    Marshaller jaxbMarshaller = jaxbContext.createMarshaller();
    jaxbMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
    jaxbMarshaller.marshal(listToSave, file);
    jaxbMarshaller.marshal(listToSave, System.out);
} catch (JAXBException e) {
    e.printStackTrace();
}
```

# Ejemplos en JAXB

- El archivo XML salvado contiene:

```
<listPersonWrapper>
  <Person>
    <fullName>John Doe</fullName>
    <id>100</id>
    <pathImage>/images/Lloroso.png</pathImage>
    <residences>
      <city>Museros</city>
      <province>Valencia</province>
    </residences>
    <residences>
      <city>Roquetas</city>
      <province>Almería</province>
    </residences>
  </Person>
</listPersonWrapper>
```



```
<Person>
  <fullName>Jane Doe</fullName>
  <id>101</id>
  <pathImage>/images/Sonriente.png</pathImage>
  <residences>
    <city>Bétera</city>
    <province>Valencia</province>
  </residences>
  <residences>
    <city>Las Negras</city>
    <province>Almería</province>
  </residences>
</Person>
</listPersonWrapper>
```

# Referencias

## **ListView Oracle**

[https://docs.oracle.com/javafx/2/ui\\_controls/list-view.htm](https://docs.oracle.com/javafx/2/ui_controls/list-view.htm)

## **Controles UI JavaFX Oracle**

[https://docs.oracle.com/javafx/2/ui\\_controls/overview.htm](https://docs.oracle.com/javafx/2/ui_controls/overview.htm)

## **XML en Wikipedia**

[http://es.wikipedia.org/wiki/Extensible\\_Markup\\_Language](http://es.wikipedia.org/wiki/Extensible_Markup_Language)

## **Introducción a XML en w3schools**

[http://www.w3schools.com/xml/xml\\_what\\_is.asp](http://www.w3schools.com/xml/xml_what_is.asp)