



P3. MODELOS Y VISTAS DE DATOS

Interfaces Persona Computador

Depto. Sistemas Informáticos y Computación

UPV

Índice

- Introducción
- Colecciones en JavaFX
 - ListView
 - ListView con imágenes
- Paso de parámetros a un controlador
- Ejercicio

Parte I

- Aplicaciones con varias ventanas
 - Único stage y varias escenas
 - Varios stages con la correspondiente escena
- Componentes gráficos adicionales
 - TableView
 - TableView con imágenes
- Ejercicio
- Anexo. Binding de propiedades

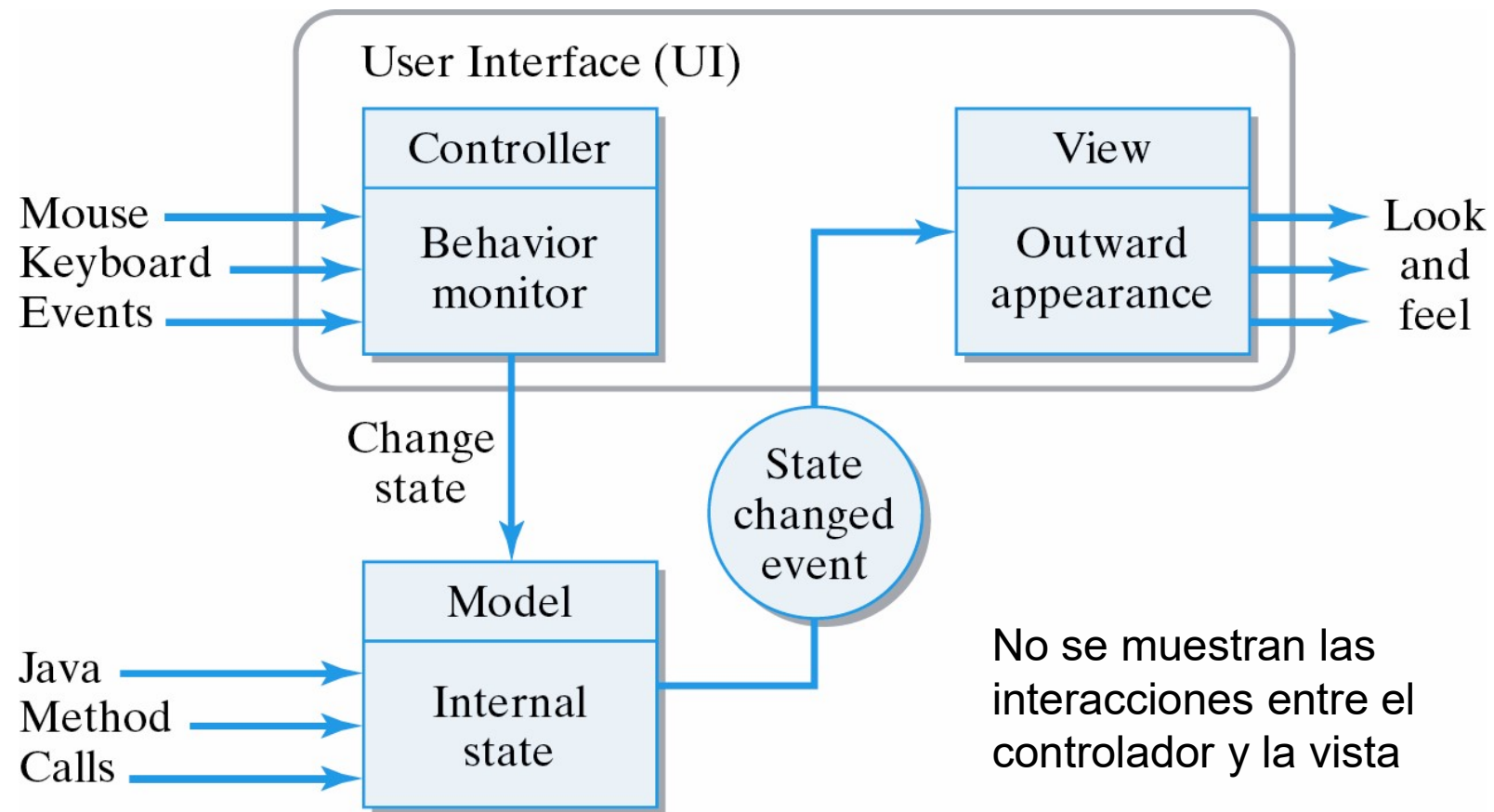
Parte II

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)

Introducción

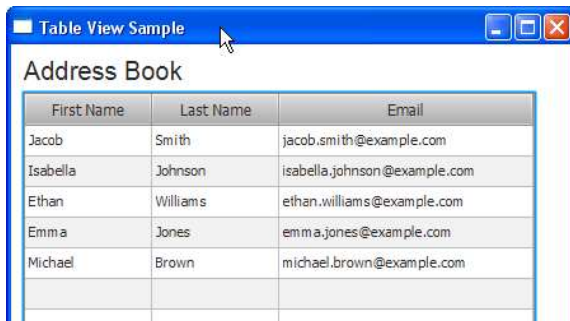
- Relaciones en la arquitectura



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 JavaFX

- Además de las colecciones habituales de Java, JavaFX introduce nuevas: `ObservableList`, `ObservableMap`
- Interfaces
 - `ObservableList`: Una lista observable 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

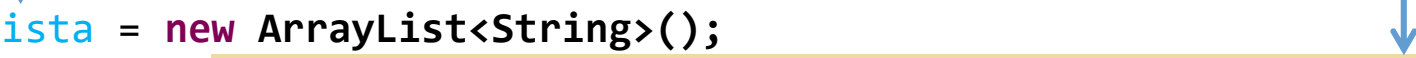
- La colección observable es una clase **envoltorio** de la lista



- 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

- **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 **listaObservable**

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



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());
```



The screenshot shows the output of a Java application. The first line is "<terminated> Main (1) [Java Application]". The subsequent lines are "Cambio detectado!", "Tamaño FX Collection: 2", and "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
```

ListView

- Permite visualizar una lista de objetos. De cada objeto se puede visualizar un texto (propiedad **text**) y una imagen (propiedad **graphic**).
- ListView tiene la propiedad **items** donde se guarda la lista de objetos

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

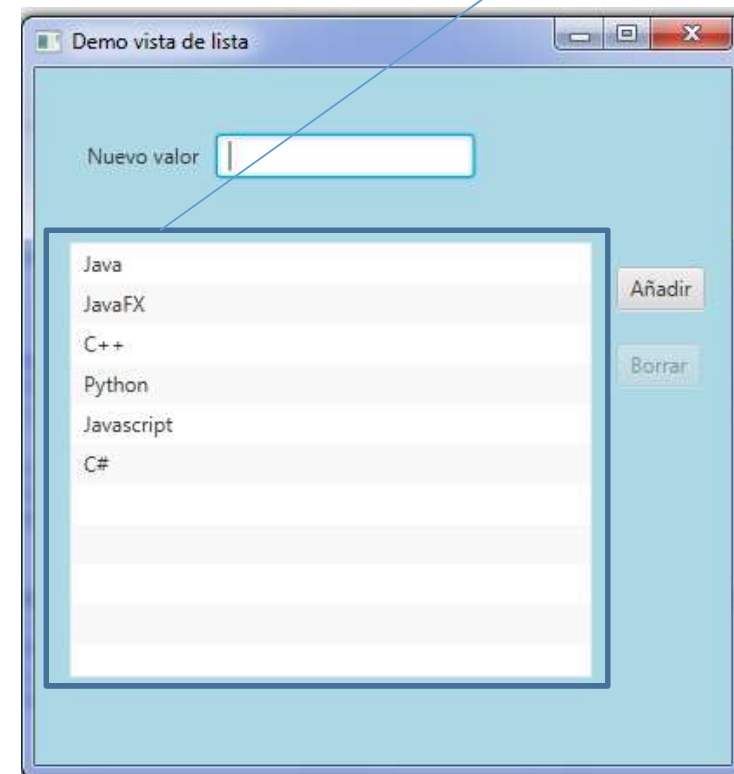
```
private ObservableList<String> datos = null;  
...  
datos = FXCollections.observableArrayList(misdatos);
```

Vinculado a la vista

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

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

ListView



ListView

- Podemos utilizar la lista observable que ya está creada en **items**, el siguiente código produce el mismo efecto que en la transparencia anterior

Clase envoltorio

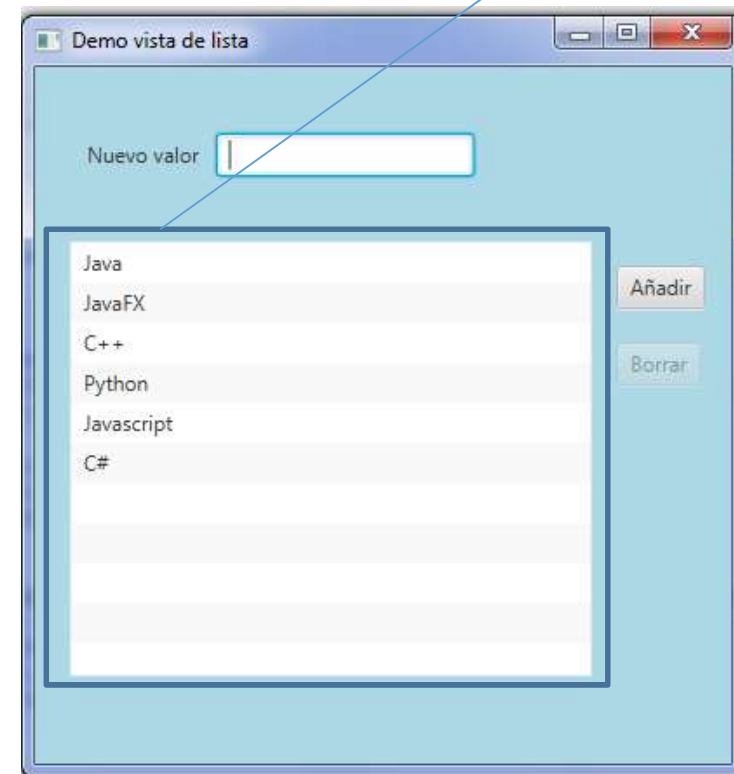
```
private ObservableList<String> datos = null;  
...  
datos = listView.getItems();
```

Datos a visualizar

```
datos.add("Java"); datos.add("JavaFX");  
datos.add("C++");  
datos.add("Python"); datos.add("Javascript");  
datos.add("C#");
```

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

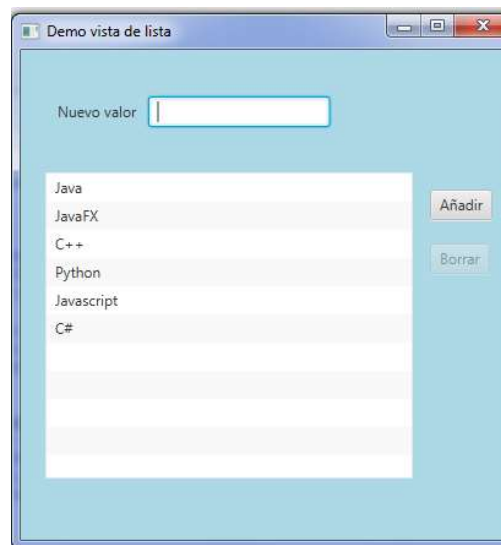
ListView



ListView

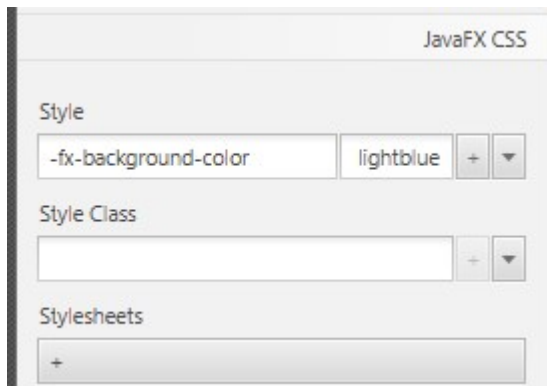
- Tanto la lista observable de objetos de la propiedad **items** como el ListView tiene que definir del mismo tipo de objeto

```
@FXML  
ListView<String> listView;  
  
private ObservableList<String> datos = null;  
...  
datos = listView.getItems();
```



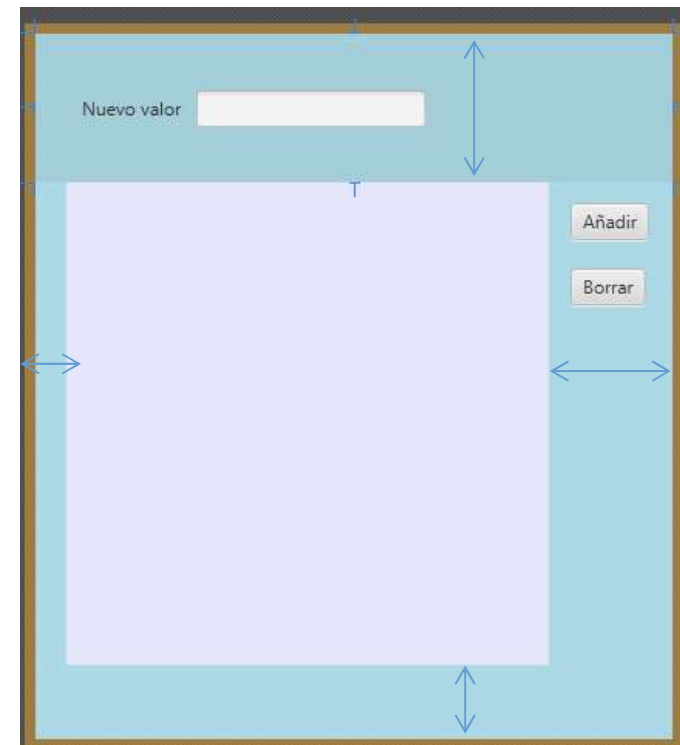
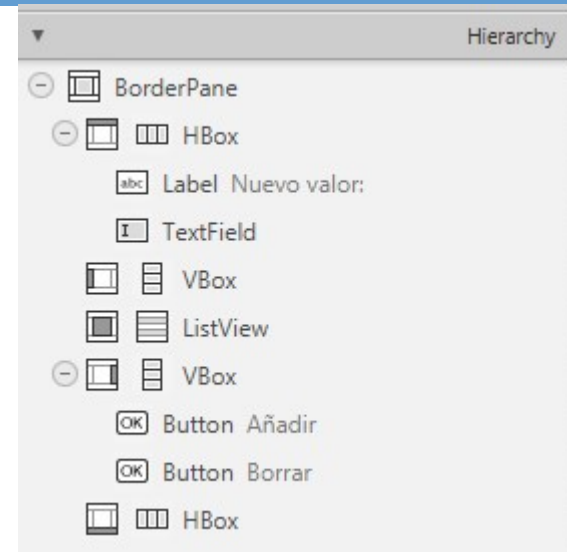
Ejemplo ListView

- Diseño de la interfaz: BorderPane, con Hbox arriba y abajo y VBox en los laterales.
- Color desde SceneBuilder con hojas de estilo CSS



- Equivalente a poner en el controlador:

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



ListView

- Métodos para obtener el objeto seleccionado en el 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

Ejemplo ListView

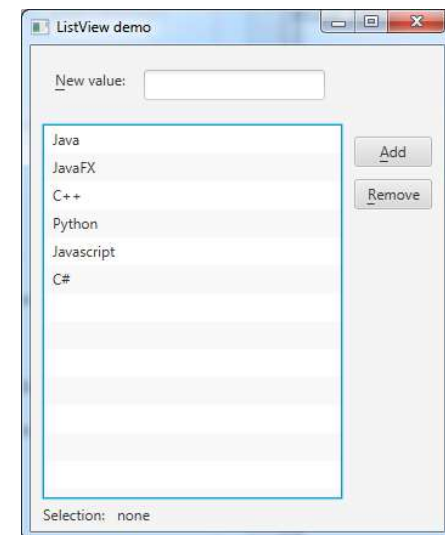
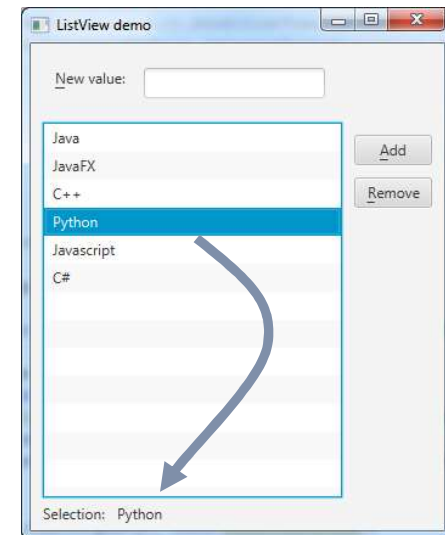
- Como detectar cambios en la selección

- Opción 1

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

- Opción 2

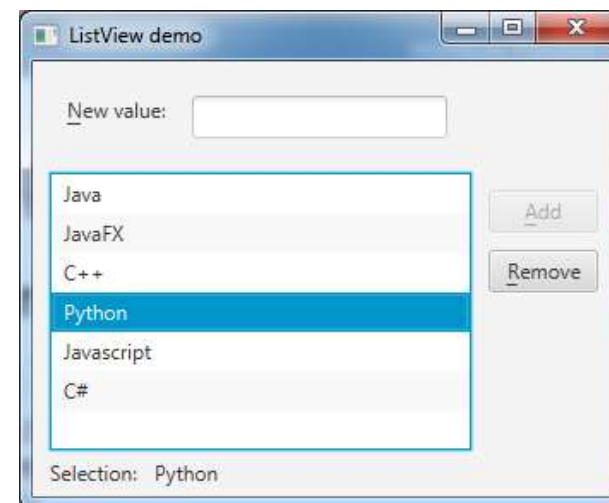
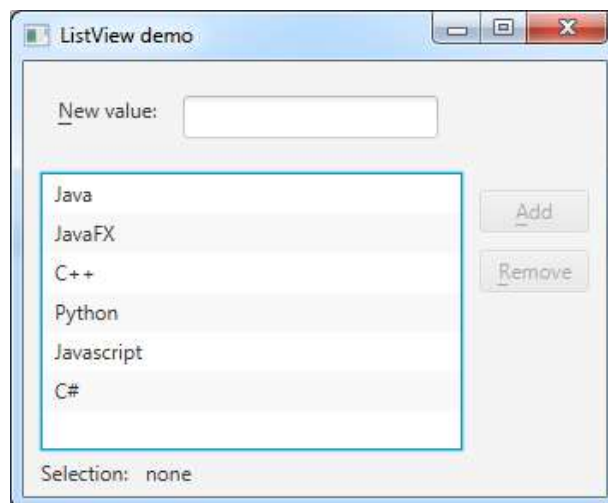
```
listView.getSelectionModel().selectedIndexProperty().  
addListener( (o, oldVal, newVal) -> {  
    if (newVal.intValue() == -1)  
        labelSelectedItem.setText("none");  
    else  
        labelSelectedItem.setText(  
            data.get(newVal.intValue()));  
});  
labelSelectedItem.setText("none");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



Ejemplo ListView

- Como detectar cambios en la selección
- Opción 3

```
labelSelectedItem.textProperty().bind(  
    Bindings.when(listView.getSelectionModel().selectedIndexProperty().isEqualTo(-1)).  
    then("none").  
    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
buttonRemove.disableProperty().bind(
    Bindings.equal(-1,
        listView.getSelectionModel().selectedIndexProperty()));
```

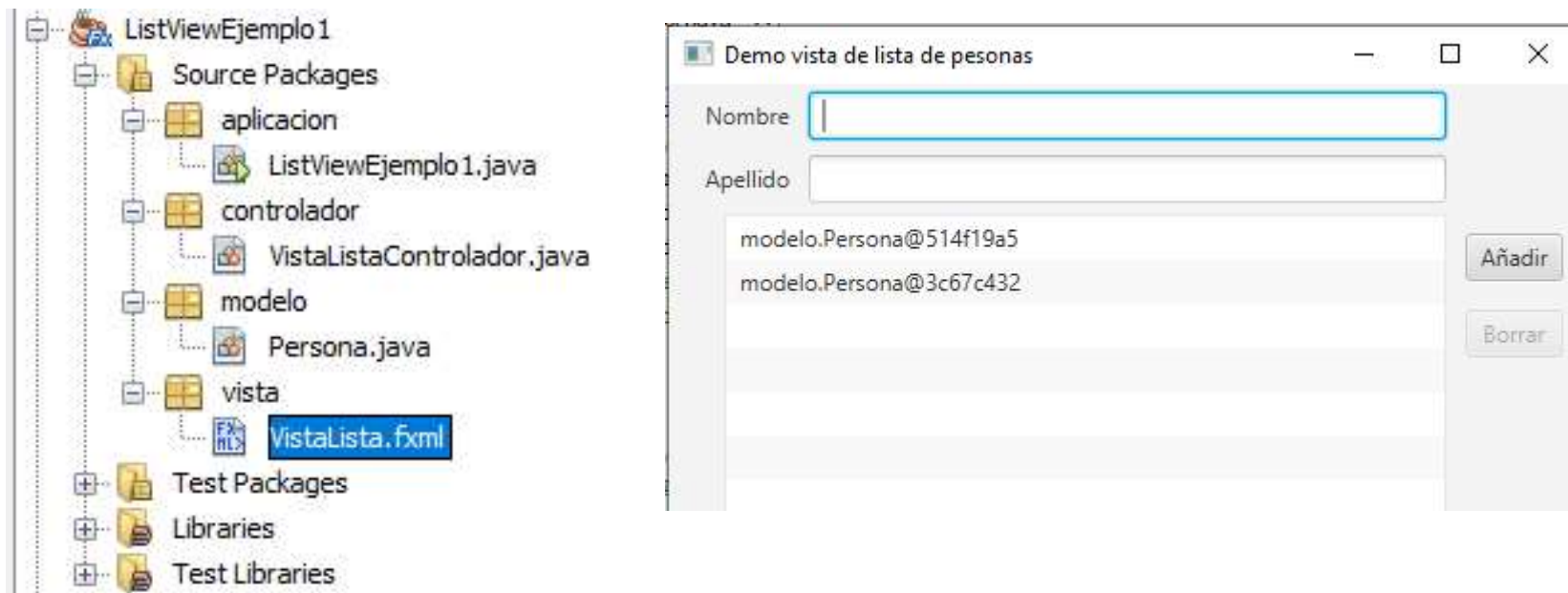
```
// The Add button will be enabled only then the TextField is not empty
buttonAdd.disableProperty().bind(firstNameText.textProperty().isEmpty())
;
```

- Los botones también puede activarse/desactivarse manualmente mediante:

```
buttonAdd.setDisable(true);
buttonRemove.setDisable(false);
```

Ejemplo ListView

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



- Observe la descomposición en paquetes del proyecto.

ListView- formato de la celda

- ListView contiene una visualización por defecto en formato texto, si se recibe un objeto se ejecuta el método `toString`.
- Cuando necesitemos una visualización particular crearemos una clase con nuestro formato, se emplean las clases `Cell` y `CellFactory`
- Todo esto es aplicable a los componentes:
 - ComboBox
 - TableView
 - TreeTableView



ListView: Cell y CellFactory

- Si queremos visualizar objetos de la clase **Lenguaje**

```
// 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());
        }
    }
}
```

- En el initialize del controlador:

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



```
package modelo;
```

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

Ejemplo 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<Persona>
{
    @Override
    protected void updateItem(Persona item, boolean empty)
    { super.updateItem(item, empty); // Obligatoria esta llamada
      if (item==null || empty) setText(null);
      else setText(item.getNombre() + " ," + item.getApellidos() );
    }
}
```

La información en pantalla

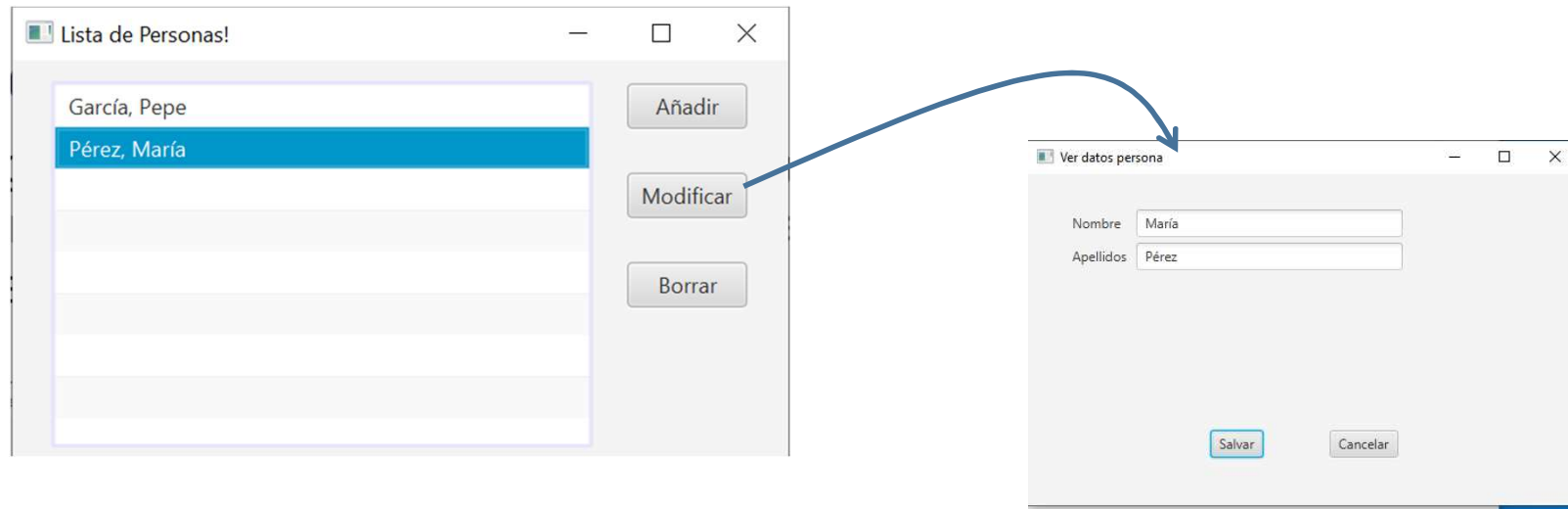
- En el initialize del controlador fijo la factoría de celdas

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



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 para editarla



```
public class DatosPersonaControlador implements Initializable {
```

```
    @FXML private TextField nombrefxID;
```

```
    @FXML private TextField apellidosfxID;
```

```
    public void initPersona( Person p)
```

```
    {
```

```
        nombrefxID.setText(p.getFirstName());
```

```
        apellidosfxID.setText(p.getLastName());
```

```
    }
```

```
}
```

Controlador de la interfaz Ver datos
persona

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()`

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

```
// acceso al controlador de datos persona
```

```
DatosPersonaControlador controladorPersona = miCargador.getController();
```

```
// obtiene los datos de la fila de la tabla seleccionada.
```

```
Persona persona = VistaTablafxID.getSelectionModel().getSelectedItem();
```

```
if (persona==null) return; // si no hay selección sale del método
```

```
controladorPersona.initPersona(persona); // pasa los datos al segundo controlador
```

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

```
Stage stage = new Stage();
```

```
stage.setScene(scene);
```

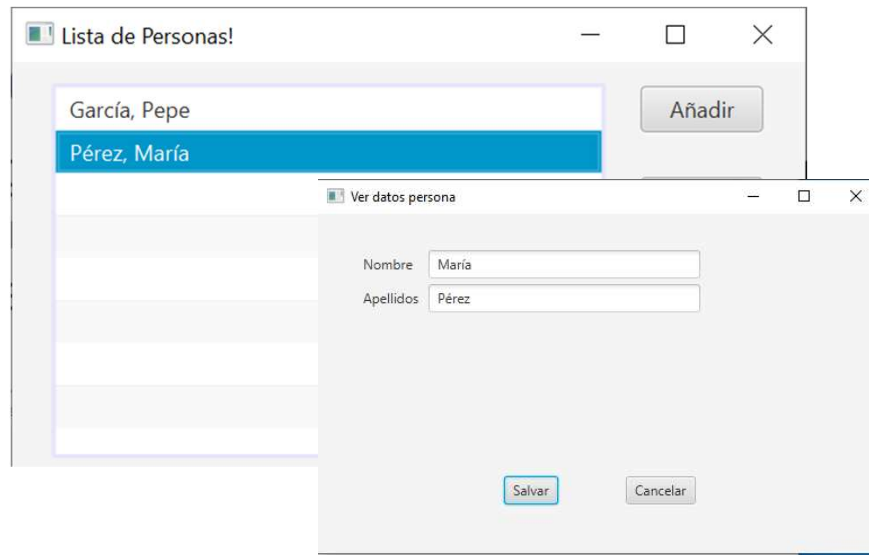
```
stage.setTitle("Ver datos persona");
```

```
stage.show();
```

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

Paso de datos a un controlador

- Tal como está el código de la transparencia anterior se mostrarían dos ventanas. La segunda no es modal.



Añade modalidad

```
Scene scene = new Scene(root,500,300);
Stage stage = new Stage();
stage.setScene(scene);
stage.setTitle("Ver datos persona");
stage.initModality(Modality.APPLICATION_MODAL);
//la ventana se muestra modal
stage.show();
```

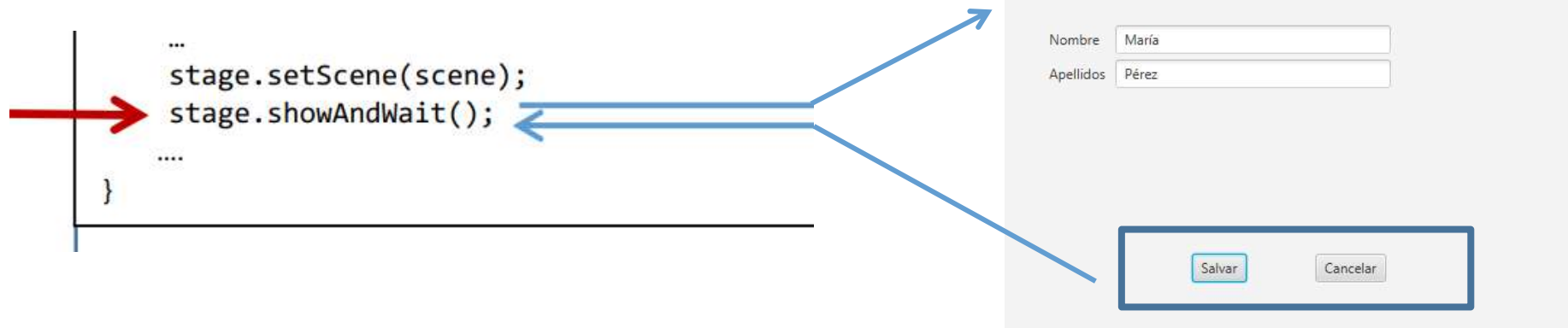
Paso de datos a un controlador

- Si queremos esperar a que la segunda ventana termine y recoger las modificaciones de la misma (nombre y/o apellidos) usaremos el método `showAndWait()`

```
@FXML
private void pulsadoModificarPersona(ActionEvent event) throws IOException {
    // Abre la ventana que muestra la información de una persona.
    FXMLLoader miCargador = new FXMLLoader(getClass().getResource("/vista/VistaPersona.fxml"));
    Parent root = miCargador.load();
    // acceso al controlador de datos persona
    DatosPersonaControlador controladorPersona = miCargador.<DatosPersonaControlador>getController();
    // fila seleccionada de la vista de tabla.
    Persona persona = VistaTablafxID.getSelectionModel().getSelectedItem();
    // persona seleccionada en la tabla
    if (persona==null)return;
    controladorPersona.initPersona(persona);
    Scene scene = new Scene(root,500,300);
    Stage stage = new Stage();
    stage.setScene(scene);
    stage.setTitle("Ver datos persona");
    stage.initModality(Modality.APPLICATION_MODAL); //la ventana se muestra modal
    stage.showAndWait(); // espera a que se cierre la segunda ventana.
}
```

Paso de datos a un controlador

- `showAndWait()` inicia un segundo hilo de eventos anidado con el primero.



- Cuando la segunda ventana se cierra el control vuelve a la primera. Podemos añadir un método al controlador de la ventana ver datos persona para obtener las modificaciones hechas en la ventana emergente.

Paso de datos a un controlador

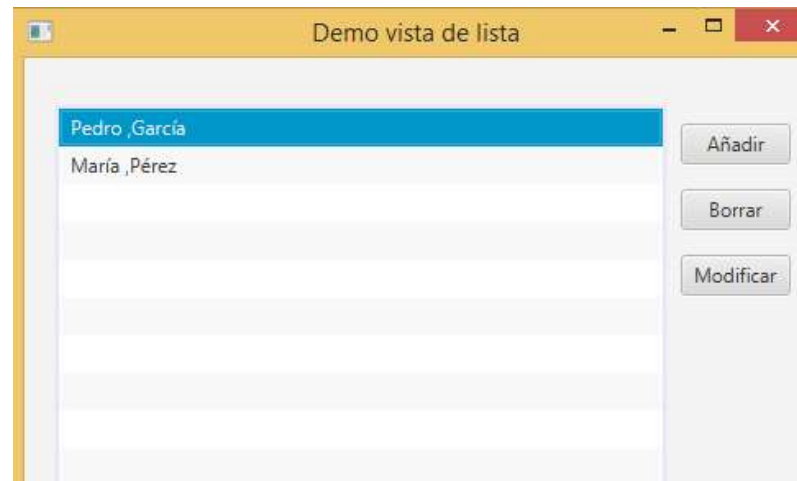
- Código para modificar

```
stage.showAndWait(); // espera a que se cierre la segunda ventana.
// para obtener el valor modificado en la ventana emergente
if (!controladorPersona.getCancelar())
{
    int indice= misPersonas.indexOf(persona);
    // índice que ocupara en la lista observable
    Persona p= controladorPersona.getPersona();
    //nuevo valor en el formulario emergente
    misPersonas.set(indice, p); // actualiza la persona.
}
```

- El método `getCancelar()` es un método publico dentro de la clase controladora que nos devolverá true en el caso de que el usuario haya cerrado la ventana tras pulsar el botón **Cancelar** o false en el caso de que se haya cerrado la ventana tras pulsar en **Salvar**

Actividad

- A partir del proyecto de la ListView con la clase Persona:
 - Crear una nueva vista con los campos Nombre y Apellido. Borrar dichos campos de la ventana original.
 - Hacer que el botón Añadir esté siempre habilitado.
 - Añadir un botón Modificar.
 - Al pulsar el botón Modificar o Añadir debe mostrarse la otra ventana para que en un caso se modifiquen los datos y en el otro se añadan.



Referencias

ListView Oracle

<https://openjfx.io/javadoc/11/javafx.controls/javafx/scene/control/ListView.html>

Controles UI JavaFX Oracle

https://docs.oracle.com/javafx/2/ui_controls/overview.htm