



P3. MODELS AND DATA VIEWS

VIRTUAL TEACHING

Purpose:

Provision of the higher education public service
(art. 1 LOU)

Responsible:

Universitat Politècnica de València.

**Rights of access, rectification, deletion,
portability, limitation or opposition to the
treatment in accordance with privacy
policies:**

<http://www.upv.es/contenidos/DPD/>

Intellectual property:

Exclusive use in the virtual classroom
environment.

Dissemination, distribution or disclosure of
recorded classes, particularly on social networks
or services dedicated to sharing notes, is
prohibited.

Violation of this prohibition may generate
disciplinary, administrative or civil liability.



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA



Interfaces Persona
Computador

Depto. Sistemas
Informáticos y Computación
UPV

Outline

- Introduction
- Collections in JavaFX
 - ListView
 - ListView with images
- Passing parameters to a controller
- Applications with multiple windows
 - One stage and several scenes
 - Several stages with their scenes
- Exercise
- Additional graphical components

Part I

- TableView
 - TableView with images
- Exercise
- Annex I: Binding of properties

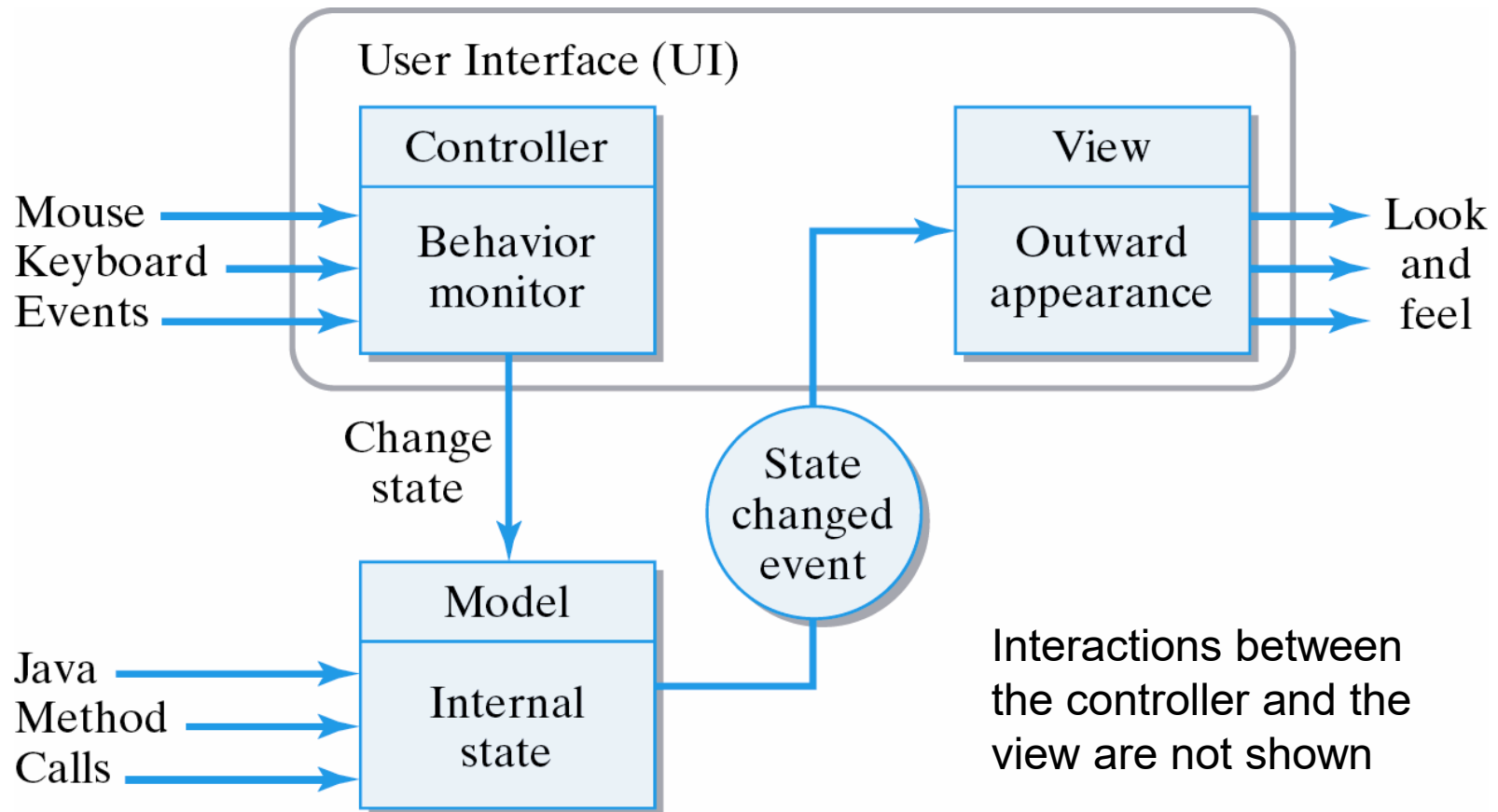
Part II

Introduction

- As mentioned in previous sessions, modern GUI applications are usually structured following the MVC pattern (Model-View-Controller)
- The architecture divides the system in 3 different parts:
 - *View*: Describes how the information is displayed
 - *Model*: Contains the state of the application, and the data it manages
 - *Controller*: What user input is accepted and what does it do with them?
- The MVC architecture was first used in *Smalltalk-80*, developed during the 70s
 - In Smalltalk, MVC was used as a model of architecture at the application-level: data (model) becomes independent from the UI (*view* and *controller*)

Introduction

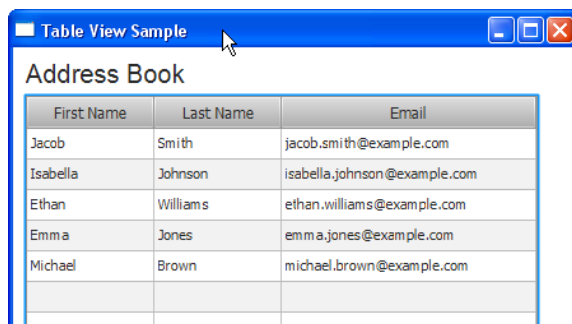
- Relationships



Introduction

- JavaFX offers specific widgets for presenting data in the user interface:
 - `ComboBox<T>`, `ListView<T>`, `TableView<T>`, `TreeTableView<T>`
- The definition of the component (view) and the data they present (model) are separated
- The model is wrapped around **observable lists**

View



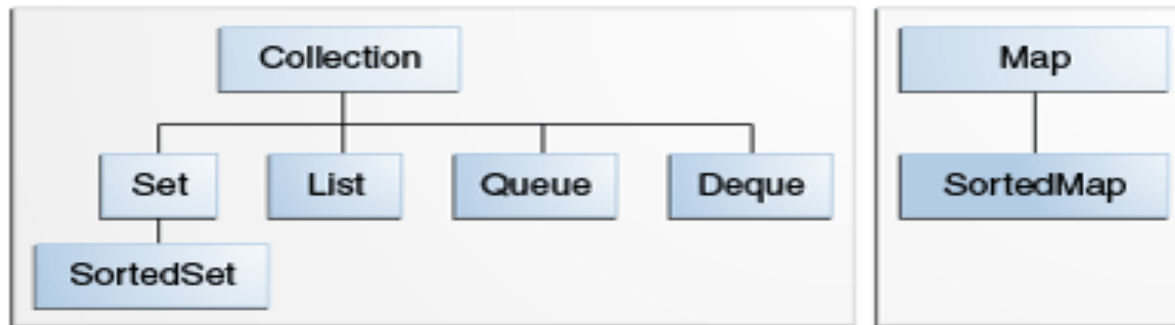
First Name	Last Name	Email
Jacob	Smith	jacob.smith@example.com
Isabella	Johnson	isabella.johnson@example.com
Ethan	Williams	ethan.williams@example.com
Emma	Jones	emma.jones@example.com
Michael	Brown	michael.brown@example.com

Model

```
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") );
```

Collections in Java

- Java collections are based on the following set of interfaces:



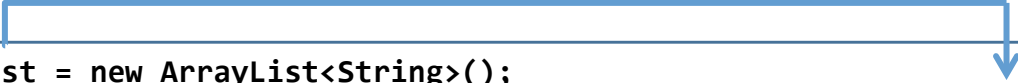
Interface	Hash	Array	Tree	Linked list	Hash+ Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Collections in JavaFX

- Besides the standard Java collections, JavaFX introduces two new interfaces: `ObservableList`, `ObservableMap`
- Interfaces
 - `ObservableList`: A list that notifies listeners whenever it changes
 - `ListChangeListener`: An interface for being able to receive change notifications from an `ObservableList`
 - `ObservableMap`: A map that notifies listeners whenever it changes
 - `MapChangeListener`: An interface for being able to receive change notifications from an `ObservableMap`

Collections in JavaFX

- FXCollections: contains static methods for wrapping Java collections into a JavaFX observable, or for creating them directly:



```
List<String> list = new ArrayList<String>();
ObservableList<String> observableList = FXMLCollections.observableList(list);

observableList.add("item one");
list.add("item two");
System.out.println("Size FX Collection: " + observableList.size());
System.out.println("Size list: " + list.size());
```

- The previous code shows:

```
Size FX Collection: 2
Size list:         2
```
- The items added through the list are visible from the FXCollection

Collections in JavaFX

- The observable collection is a wrapper around the list



- The listeners of the JavaFX collections are only notified when changes on the list are made through the `observableList`

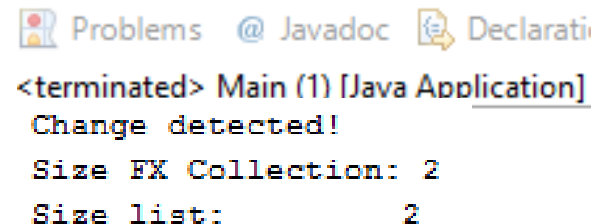
Collections in JavaFX

- A listener can be registered in the observable list to receive notifications of changes:

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

- Running this code results in:

```
observableList.add("item one");  
list.add("item two");  
System.out.println("Size FX Collection: " + observableList.size());  
System.out.println("Size list: " + list.size());
```




```
<terminated> Main (1) [Java Application]  
Change detected!  
Size FX Collection: 2  
Size list: 2
```

Collections in JavaFX

- The `ListChangeListener.Change` parameter provides information about the change:

```
observableList.addListener(new ListChangeListener<String>() {  
    @Override  
    public void onChanged(ListChangeListener.Change<? extends String> arg0) {  
        System.out.println("Change detected!");  
        while(arg0.next())  
        { System.out.println("Added? " + arg0.wasAdded());  
          System.out.println("Removed? " + arg0.wasRemoved());  
          System.out.println("Permutated? " + arg0.wasPermutated());  
          System.out.println("Replaced? " + arg0.wasReplaced());  
        }  
    }  
});
```

```
...  
observableList.add("item one");  
list.add("item two");
```



```
Change detected!  
Added? true  
Removed? false  
Permutated? false  
Replaced? false  
Size FX Collection: 2  
Size list:          2
```

Example of ListView

- FX Collections are used to define the model for some graphical components.
- ListView

Show this data

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

Wrapper class

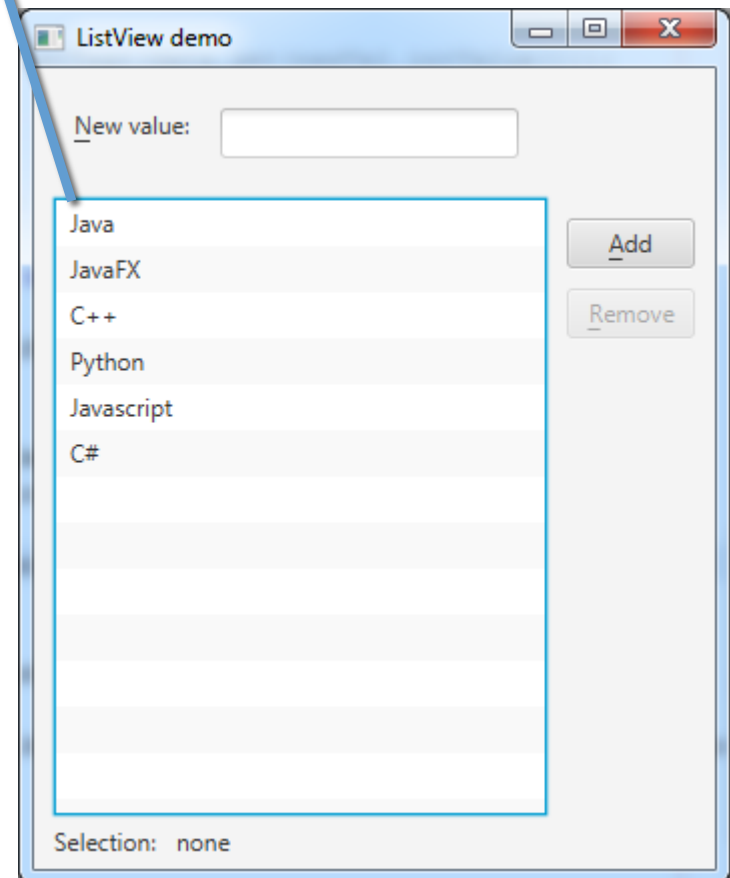
```
private ObservableList<String> data = null;  
...  
data = FXCollections.observableArrayList(myData);
```

Bind to listview

```
listView.setItems(data);
```

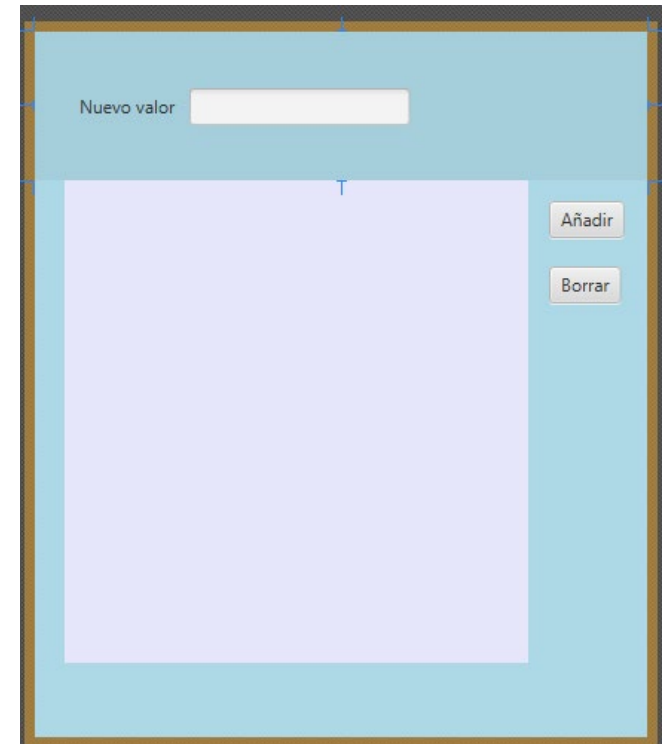
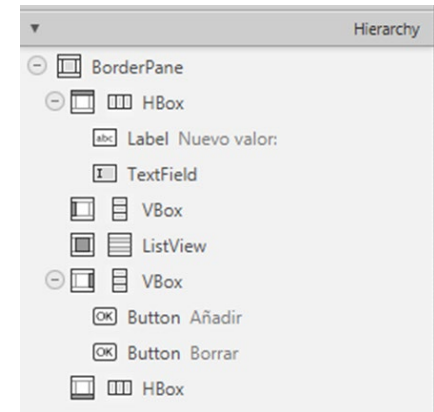
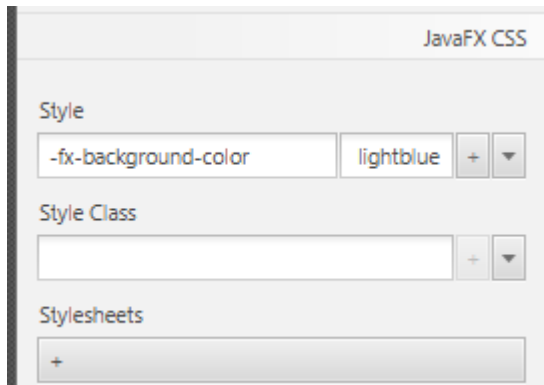
Changes in the observable list are automatically reflected in the list view: add, remove.

ListView



Example of ListView

- Interface design: BorderPane with Hboxes in the top and bottom, and two VBoxes in the left and right borders
- Color: using the property CSS Style in Scene Builder



- Equivalent to add into the controller:

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

Example of ListView

- Useful methods in ListView:
 - `getSelectionModel().getSelectedIndex()`: if the list is in single selection mode, returns the index of the selected item
 - `getSelectionModel().getSelectedItem()`: returns the selected item
 - `getFocusModel().getFocusedIndex()`: returns the index of the focused item
 - `getFocusModel().getFocusedItem()`: returns the focused element
- The ListView can be configured in multiple selection mode
`getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);`
- The methods `getSelectedIndices()` and `getSelectedItems()` in `MultipleSelectionModel` return observable lists that can be used to monitor changes in selection

Example of ListView

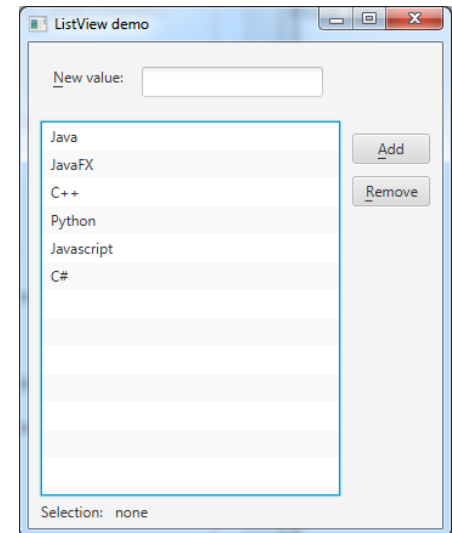
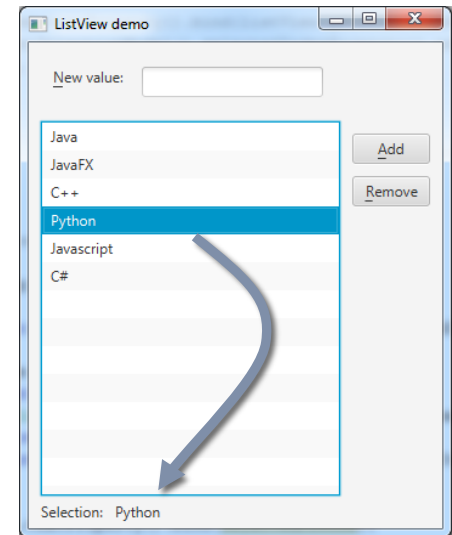
- Listening to changes in selection

- Option 1

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

- Option 2

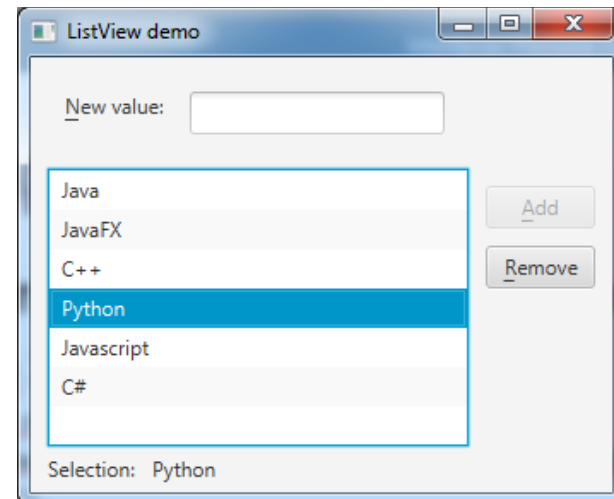
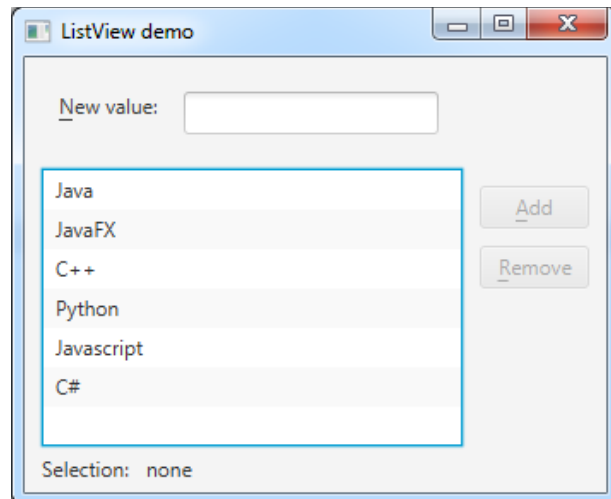
```
listView.getSelectionModel().selectedIndexProperty().  
    addListener( (o, oldVal, newVal) -> {  
        if (newVal.intValue() == -1)  
            selectedItem.setText("none");  
        else selectedItem.setText(  
            listView.getItems().get(newVal.intValue()));  
    });  
selectedItem.setText("none");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



Example of ListView

- Listening to changes in selection
- Option 3

```
selectedItem.textProperty().bind(  
    Bindings.when(listView.getSelectionModel().selectedIndexProperty().isEqualTo(-1)).  
    then("none").  
    otherwise(listView.getSelectionModel().selectedItemProperty().asString()));
```



Example of ListView

- Enabling/disabling buttons when changing the selection

// 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

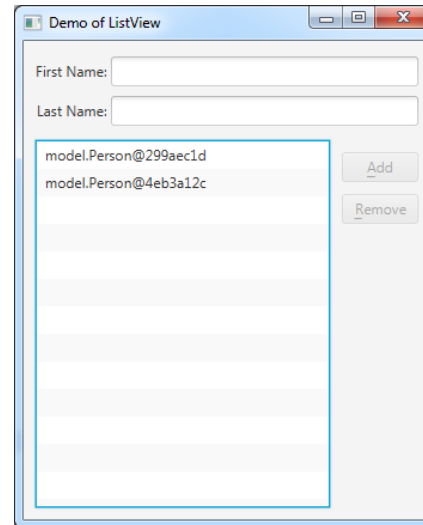
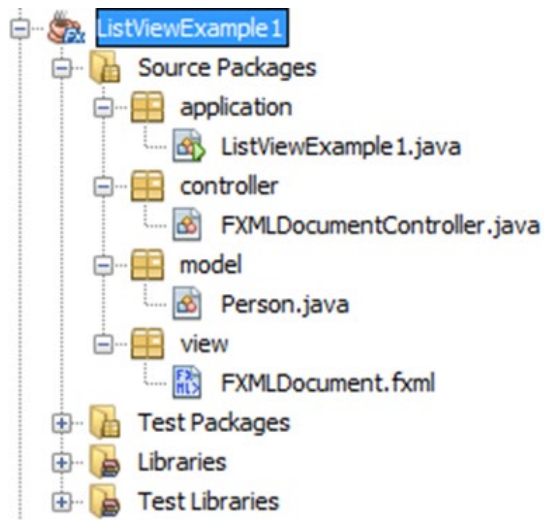
```
buttonAdd.disableProperty().bind(firstNameText.textProperty().isEmpty());
```

- Buttons can also be manually disabled/enabled with:

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

Example of ListView

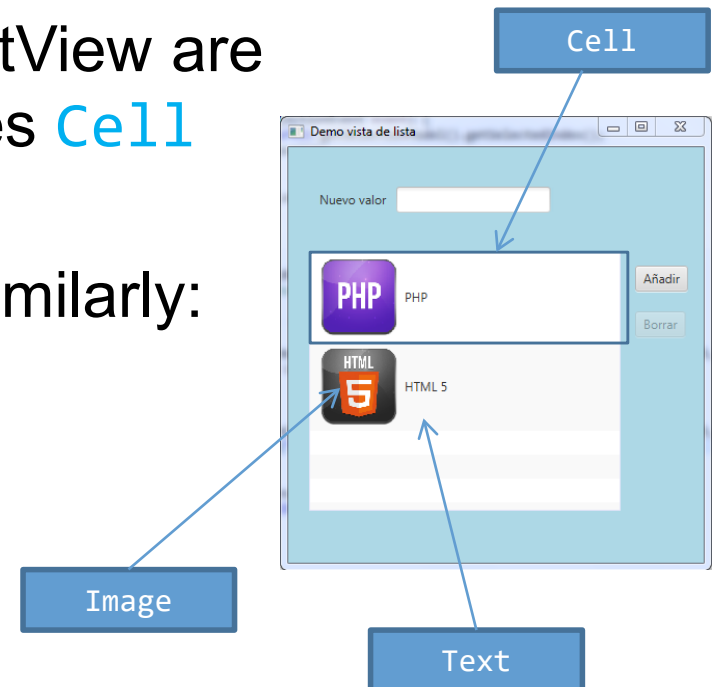
- Download the example from poliformaT and open it in NetBeans. The project will have the following structure:



- Note the organization in different packages

Example of ListView

- ListView by default shows strings
- Objects are shown calling `toString()` method of that object
- To control how the elements of a ListView are rendered, we have to use the classes **Cell** and **CellFactory**
- There are other controls that work similarly:
 - ComboBox
 - TableView
 - TreeTableView



ListView: Cell and CellFactory

- The ListCell indicates how to show a *Person* in the ListView.

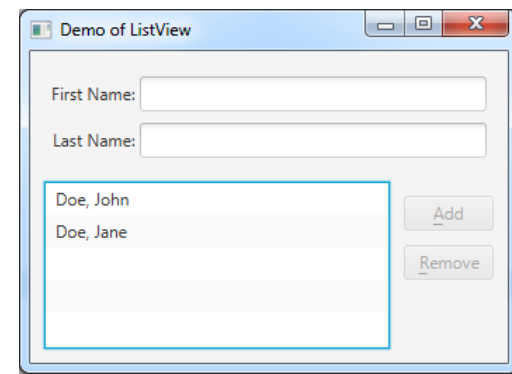
The class in charge of showing a cell

```
// Local class in the controller
class PersonListCell extends ListCell<Person>
{
    @Override
    protected void updateItem(Person item, boolean empty)
    { super.updateItem(item, empty); // This is mandatory
      if (item==null || empty) setText(null);
      else setText(item.getLastName() + ", " + item.getFirstName());
    }
}
```

The cell's content

- Set the listView's cell factory in the method initialize in the controller

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



ListView: Cell and CellFactory

- The ListCell class also allows us to add an image to the elements of the list

```
// Local class to the controller
class LanguageListCell extends ListCell<Language>
{
    private ImageView view = new ImageView();
    @Override
    protected void updateItem(Language item, boolean empty)
    {
        super.updateItem(item, empty);
        if (item==null || empty) {
            setText(null);
            setGraphic(null);}
        else {
            view.setImage(item.getImage());
            setGraphic(view);
            setText(item.getName());
        }
    }
}
```



```
package model;
```

```
public class Language {
    private String name;
    private Image image;
```

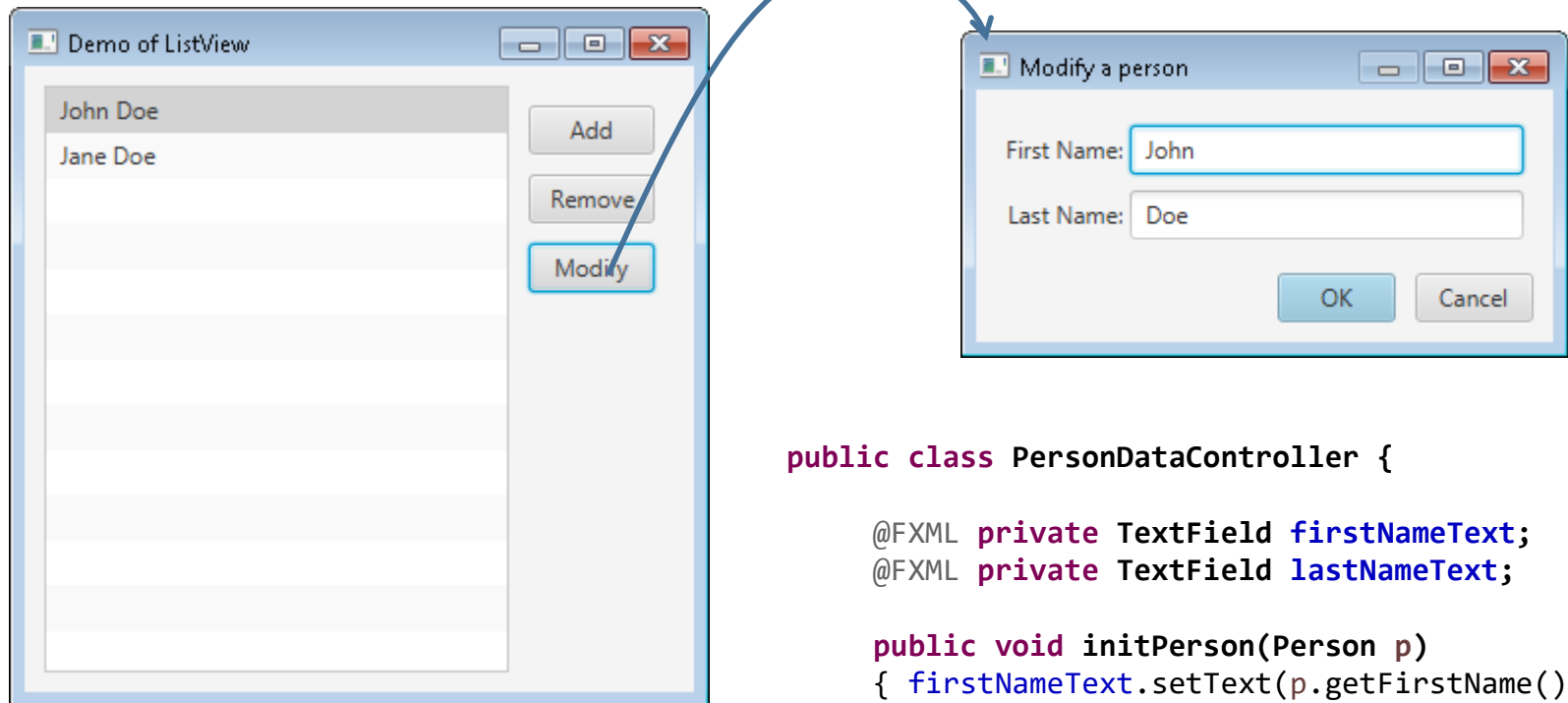
```
...
```

- In the method initialize() of the controller:

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

Passing data to controllers

- Suppose we use a form to show information about a person. We have to pass the Person's information as a parameter



```
public class PersonDataController {  
  
    @FXML private TextField firstNameText;  
    @FXML private TextField lastNameText;  
  
    public void initPerson(Person p)  
    { firstNameText.setText(p.getFirstName());  
      lastNameText.setText(p.getLastName());  
    }  
}
```

Passing data to controllers

- After loading the form's fxml file, we can obtain a reference to its controller and invoke the previous method (`initPerson`)

```
//AnchorPane root = (AnchorPane)FXMLLoader.load(getClass().getResource("/view/PersonData.fxml"));
```



Change to non-static call

```
FXMLLoader myLoader = new FXMLLoader(getClass().getResource("/view/PersonData.fxml"));  
AnchorPane root = (AnchorPane) myLoader.load();
```

```
// obtain a reference to the controller  
PersonDataController personController = myLoader.<PersonDataController>getController();  
personController.initPerson(person);
```

```
Scene scene = new Scene(root,400,400);  
Stage stage = new Stage();  
stage.setScene(scene);  
stage.setTitle("Person details");  
stage.show();
```

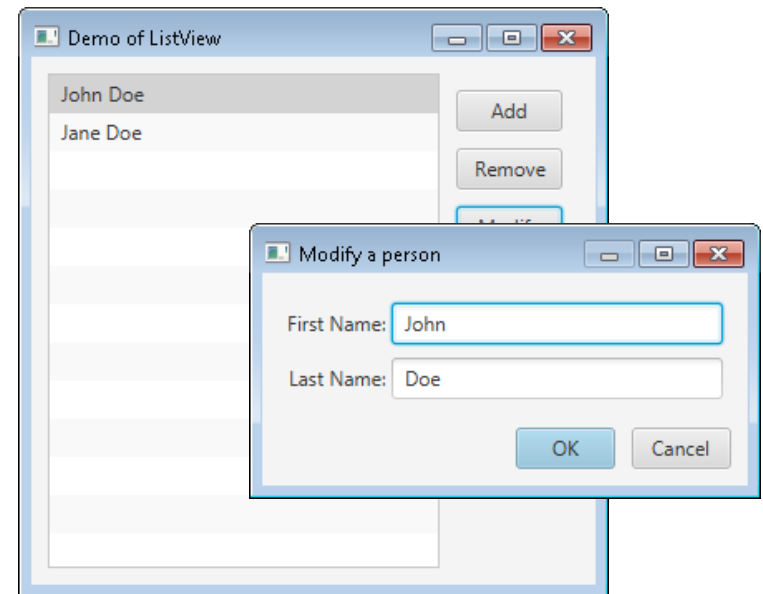
- This code is in the event handler of the button `Modify`.

Passing data to controllers

- In the previous example, the second windows is no modal
- To adding modality, we use the method: `initModality()`

```
Scene scene = new Scene(root,400,400);  
Stage stage = new Stage();  
stage.setScene(scene);  
stage.setTitle("Person details");  
stage.initModality(Modality.APPLICATION_MODAL);  
stage.show();
```

Adds modality

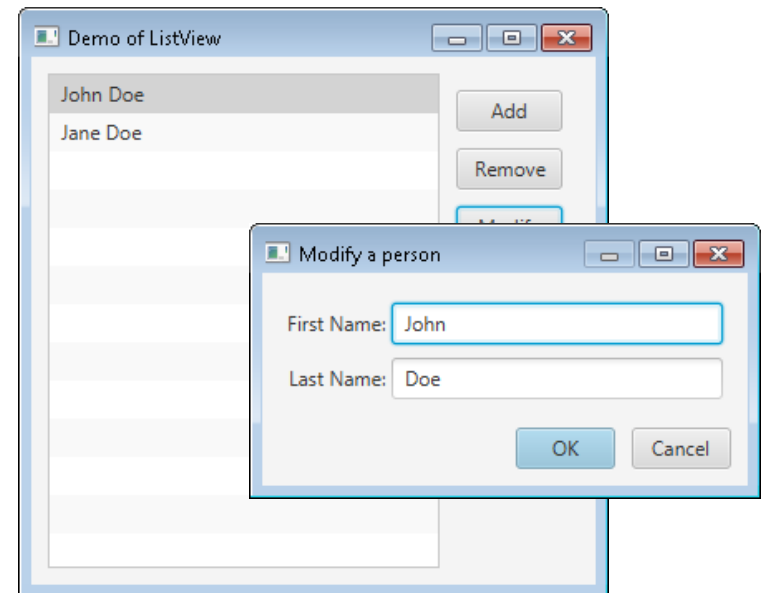


Passing data to controllers

- If the applications needs to collect the data or updates made by the second window, it must wait until it closes usind `showAndWait()`

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

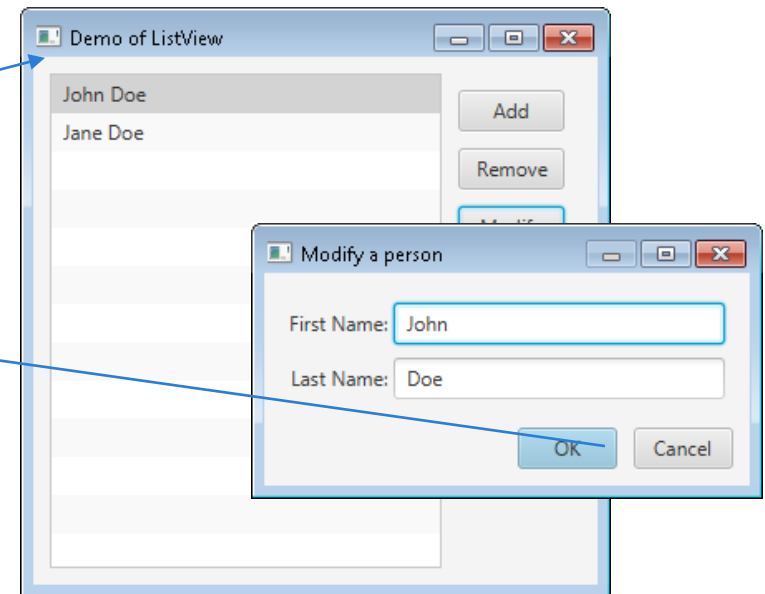
Execution continues when second window closes



Passing data to controllers

- **showAndWait():** Shows this stage and waits for it to be hidden (closed) before returning to the caller.

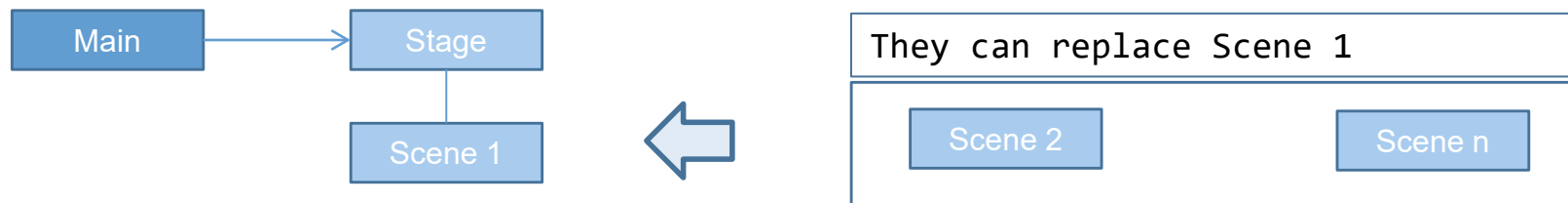
```
Scene scene = new Scene(root,400,400);
Stage stage = new Stage();
stage.setScene(scene);
stage.setTitle("Person details");
stage.initModality(Modality.APPLICATION_MODAL);
stage.showAndWait();
if(!personController.getCancelled())
{
    Person modifiedPerson = personController.getPersona();
    ...
}
```



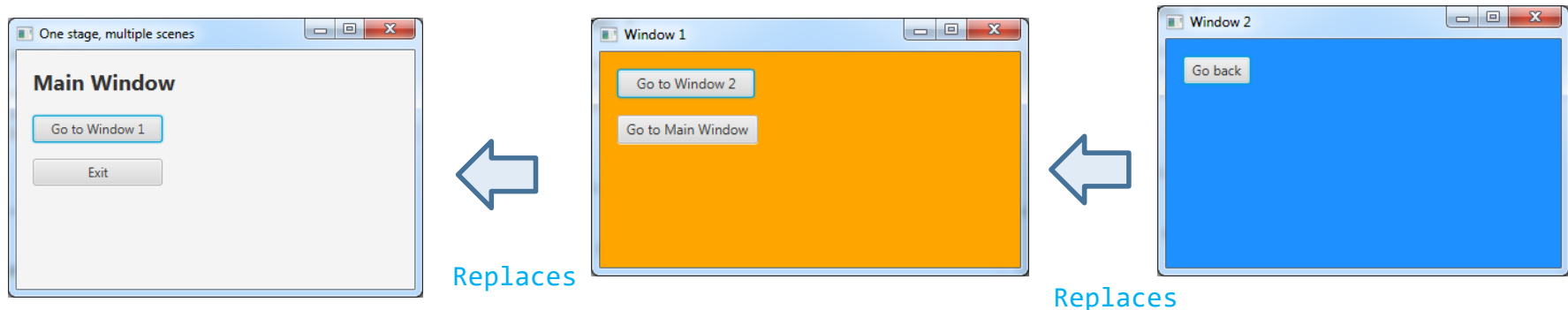
Execution continues when second window closes

Applications with multiple windows

- Applications can have a single **Stage** with multiple scenes



- The application has a single visible window (Stage)



- Each window receives the stage and, each controller loads the next scene

Multiple windows: one stage

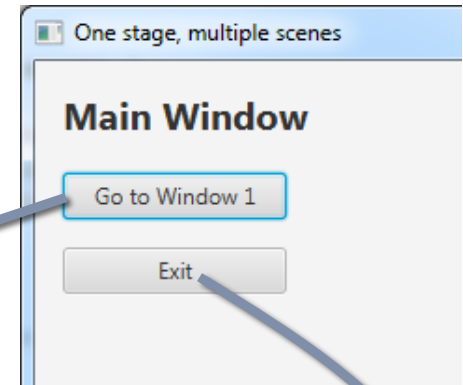
- The stage is passed as a parameter to the controller of each window

```
public class SingleStage extends Application {  
    @Override  
    public void start(Stage stage) throws Exception {  
        FXMLLoader loader =  
            new FXMLLoader(getClass().getResource("/view/MainWindow.fxml"));  
        Parent root = loader.load();  
        Scene scene = new Scene(root);  
        MainWindowController mainController =  
            loader.<MainWindowController>getController();  
        mainController.initStage(stage);  
        stage.setTitle("One stage, multiple scenes");  
        stage.setScene(scene);  
        stage.show();  
    }  
    public static void main(String[] args) {  
        launch(args);  
    }  
}
```

Multiple windows: one stage

- Controller class for the main window

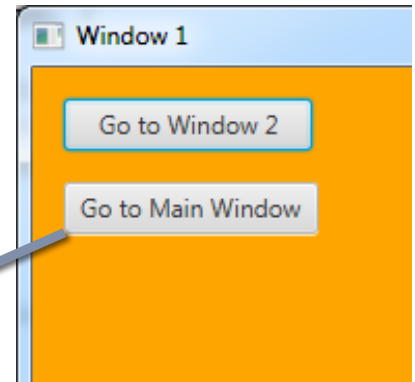
```
public class MainWindowController implements Initializable {  
    private Stage primaryStage;  
    public void initStage(Stage stage) {  
        primaryStage = stage;  
    }  
    @FXML  
    private void onGoToWindow1(ActionEvent event) {  
        try {  
            FXMLLoader myLoader = new FXMLLoader(getClass().getResource("/view/Window1.fxml"));  
            Parent root = (Parent) myLoader.load();  
            Window1Controller window1 = myLoader.<Window1Controller>getController();  
            window1.initStage(primaryStage);  
            Scene scene = new Scene(root);  
            primaryStage.setScene(scene);  
            // primaryStage.show(); // Not necessary  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
    @FXML  
    private void onExit(ActionEvent event) {  
        primaryStage.hide();  
    }  
}
```



Multiple windows: one stage

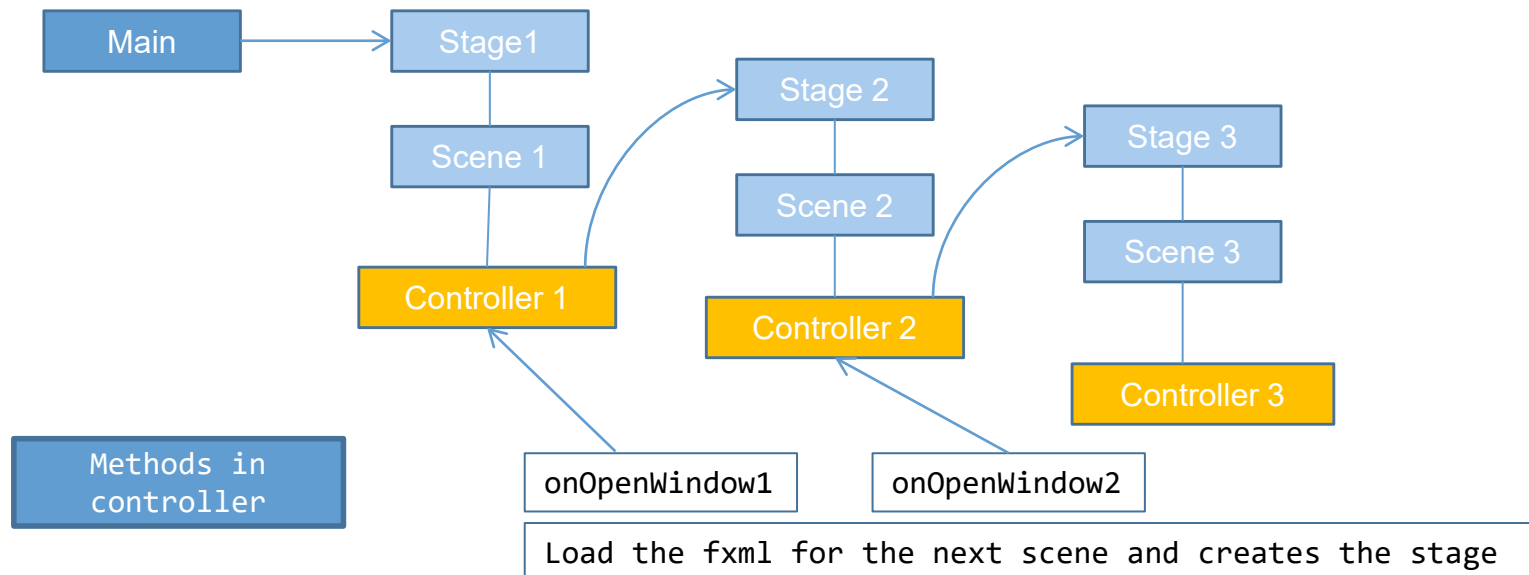
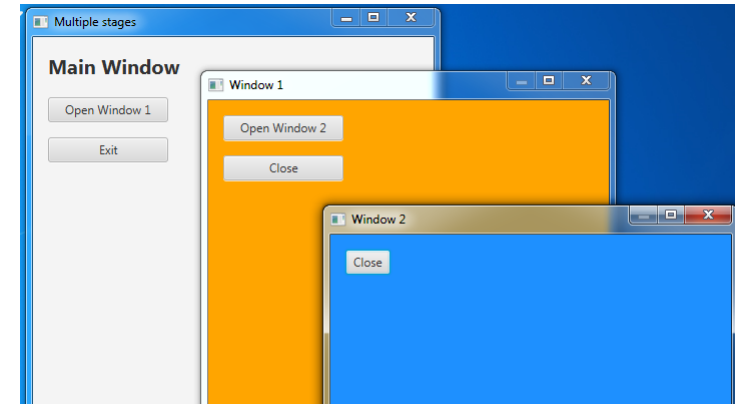
- Controller class for Window 1:

```
public class Window1Controller implements Initializable {  
    private Stage primaryStage;  
    private Scene prevScene;  
    private String prevTitle;  
  
    public void initStage(Stage stage) {  
        primaryStage = stage;  
        prevScene = stage.getScene();  
        prevTitle = stage.getTitle();  
        primaryStage.setTitle("Window 1");  
    }  
    @FXML  
    private void onGoToWindow2(ActionEvent event) {  
        // Similar to onGoToWindow1  
    }  
    @FXML  
    private void onGoToMainWindow(ActionEvent event) {  
        primaryStage.setTitle(prevTitle);  
        primaryStage.setScene(prevScene);  
    }  
}
```



Applications with multiple windows

- We can use multiple stages each with a scene
- The three windows are visible
- Defined as modal, except the main window
- Each controller loads the next stage



Applications with multiple windows

- The code of the main class is similar to the previous example
- Each scene has its own stage

```
public class MultipleStages extends Application {  
    @Override  
    public void start(Stage primaryStage) {  
        try {  
            FXMLLoader myLoader = new FXMLLoader(getClass().getResource("/view/MainWindow.fxml"));  
            Parent root = (Parent)myLoader.load();  
            Scene scene = new Scene(root, 400, 400);  
            primaryStage.setTitle("Multiple stages");  
            primaryStage.setScene(scene);  
            primaryStage.show();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
    public static void main(String[] args) {  
        launch(args);  
    }  
}
```


Applications with multiple windows

- Main Controller class

```
public class MainWindowController implements Initializable {  
    @FXML private void onOpenWindow1(ActionEvent event) {  
        try {  
            Stage aNewStage = new Stage();  
            FXMLLoader myLoader = new  
                FXMLLoader(getClass().getResource("/view/Window1.fxml"));  
            Parent root = (Parent) myLoader.load();  
            myLoader.<Window1Controller>getController().initStage(aNewStage);  
            Scene scene = new Scene(root, 400, 400);  
            aNewStage.setScene(scene);  
            aNewStage.initModality(Modality.APPLICATION_MODAL);  
            aNewStage.show();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
    @FXML private void onExit(ActionEvent event) {  
        Node n = (Node) event.getSource();  
        n.getScene().getWindow().hide();  
    }  
}
```



Window 1

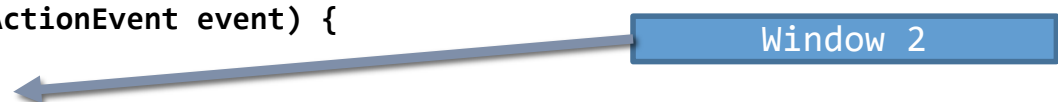


Modality

Applications with multiple windows

- Controller class for Window 1

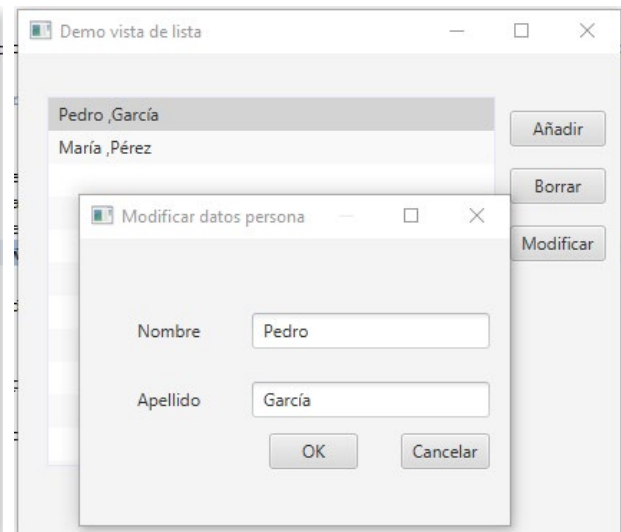
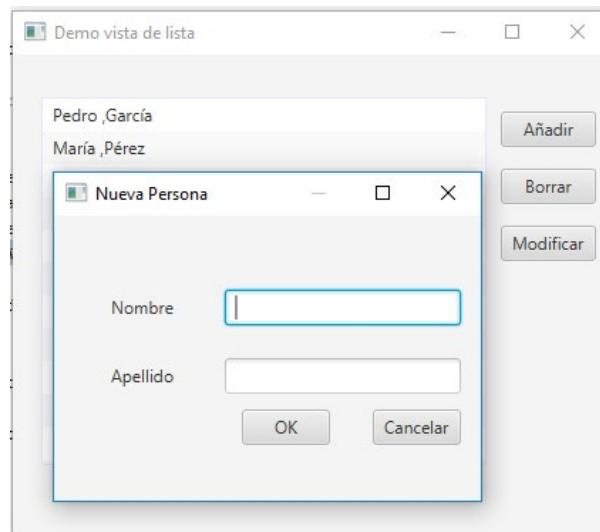
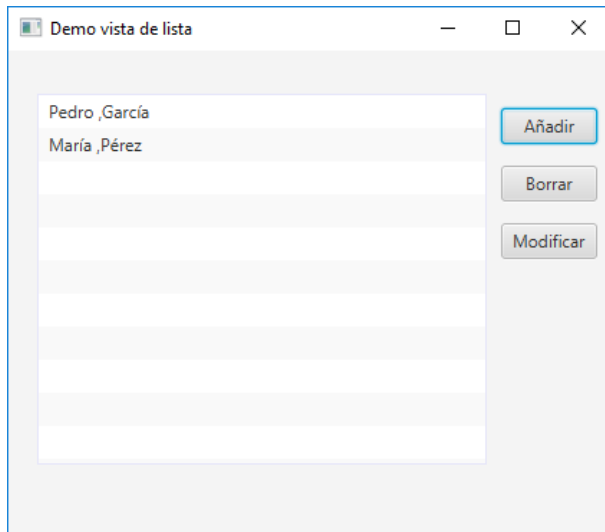
```
public class Window1Controller implements Initializable {
    private Stage myOwnStage;
    public void initStage(Stage stage) {
        myOwnStage = stage;
        myOwnStage.setTitle("Window 1");
    }
    @FXML private void onOpenWindow2(ActionEvent event) {
        try {
            Stage aNewStage = new Stage();
            FXMLLoader myLoader = new FXMLLoader(getClass().getResource("/view/Window2.fxml"));
            Parent root = (Parent) myLoader.load();
            myLoader.<Window2Controller>getController().initStage(aNewStage);
            Scene scene = new Scene(root, 400, 400);
            aNewStage.setScene(scene);
            aNewStage.initModality(Modality.APPLICATION_MODAL);
            aNewStage.show();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    @FXML private void onClose(ActionEvent event) {
        Node n = (Node) event.getSource();
        n.getScene().getWindow().hide();
    }
}
```



The diagram illustrates the relationship between the code and the UI element. A blue rectangular box labeled "Window 2" is positioned to the right of the code. A grey arrow originates from the box and points to the line `Stage aNewStage = new Stage();` within the `onOpenWindow2` method, indicating that this line of code is responsible for creating the window.

Exercise

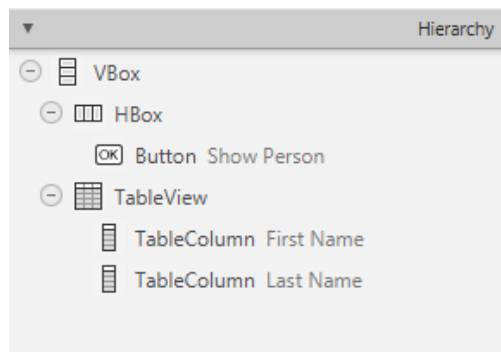
- Start with the ListView example that uses the Person class (ListViewExample1.zip):
 - Make the list to show for each person: <First name> <Last name>
 - Make the Add button to be always enabled
 - Add a Modify button (only should be enabled if some element of the list is selected)
 - After pressing the button Modify or Add, the other window should be shown for modifying the data of a existing person, or to add a new person



PART 2

TableView

- The control shows rows of data, where each row is divided into columns
- **TableColumn** represents a column in the table and contains a **CellValueFactory** for defining how to obtain the cell's content
- If the table only contains text columns:
 - Scene Builder



Assigned fx:id

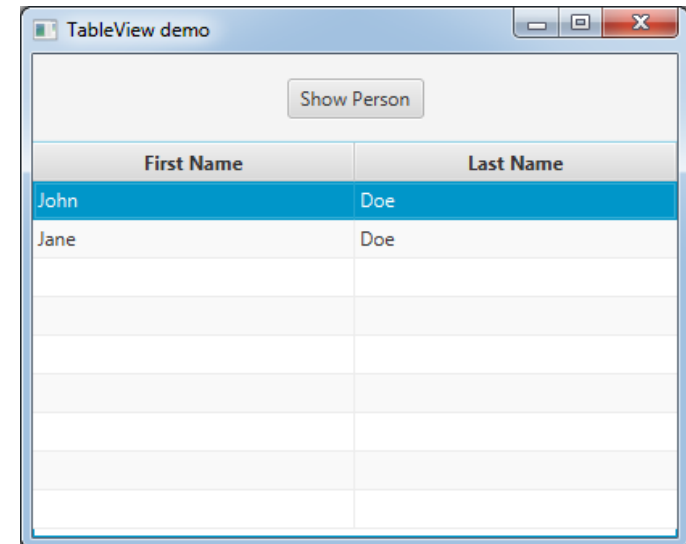
fx:id	Component
tableView	TableView
firstNameColumn	TableColumn
lastNameColumn	TableColumn

3 items



TableView

- The table contains instances of the class Person
- The columns are First name and Last Name



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

TableView

- First, we have to indicate the type of objects shown by the TableView, and the data type shown in each column
- In the controller:

```
@FXML private TableView<?> tableView;  
@FXML private TableColumn<?, ?> firstNameColumn;  
@FXML private TableColumn<?, ?> lastNameColumn;
```

- Change to:

```
@FXML private TableView<Person> tableView; // The rows' class  
@FXML private TableColumn<Person, String> firstNameColumn;  
@FXML private TableColumn<Person, String> lastNameColumn;
```



This column will
show a piece of
data from a Person

...and that piece of
data is a String

TableView

- Then we have to indicate how to compute the data shown in each column using the method `setCellValueFactory` in `TableColumn`
- In the controller's initialize method:

```
firstNameColumn.setCellValueFactory(  
    new PropertyValueFactory<Person, String>("firstName"));  
lastNameColumn.setCellValueFactory(  
    new PropertyValueFactory<Person, String>("lastName"));  
tableView.setItems(myData);
```

- The `PropertyValueFactory<Person,String>(String prop)` class:
 - Is a convenience class for extracting a property from a `Person`
 - Internally, it will try to invoke: `<prop>Property()`, `get<prop>` or `is<prop>` in the `Person` object to be shown

TableView

- The TableView contains Person instances
- The columns are the first and last names

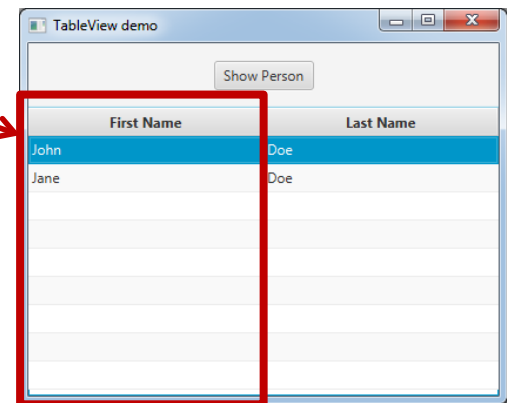
```
firstNameColumn.setCellValueFactory(  
    new PropertyValueFactory<Person, String>("firstName"));
```

```
public class Person {  
    private final StringProperty firstName = new SimpleStringProperty();  
    private final StringProperty lastName = new SimpleStringProperty();
```

```
    public Person(String firstName, String lastName) {  
        this.firstName.setValue(firstName);  
        this.lastName.setValue(lastName);  
    }  
    ...
```

```
    public StringProperty firstNameProperty() {  
        return firstName;  
    }  
}
```

```
private ObservableList<Person> myData =  
    FXCollections.observableArrayList();  
  
myData.add(new Person("John", "Doe"));  
myData.add(new Person("Jane", "Doe"));
```



TableView

CellValueFactory specifies the data assigned to a column

CellFactory specifies how data is displayed

- The code:

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

- is equivalent to:

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

RunTime exception if it doesn't exists

- Or:

```
firstNameColumn.setCellValueFactory(cellData ->  
    cellData.getValue().firstNameProperty());
```

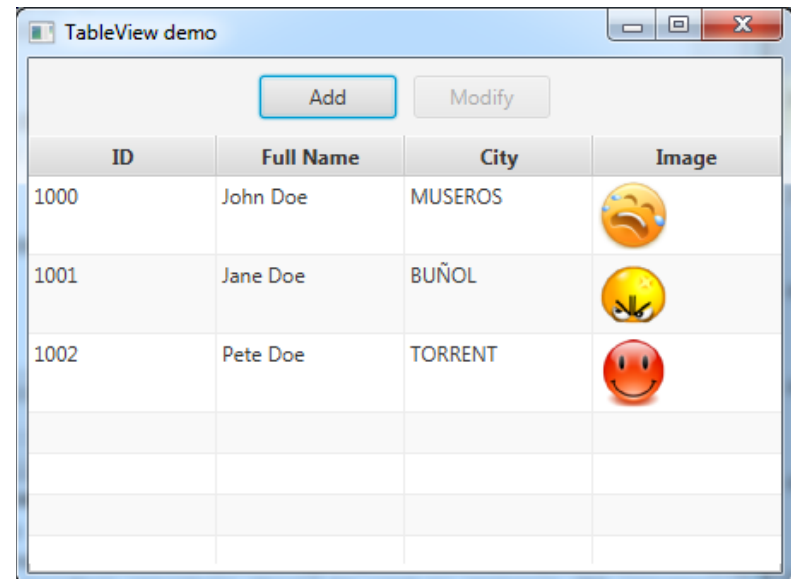
TableView with images

- Let's modify the table to show an image and a field (city) from a separate class:

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

```
// Immutable class  
public class Residence {  
    private final String city;  
    private final String province;  
    // And constructor and getters  
}
```

The Person class have the JavaFX properties, getters and setters. NetBeans hint: right click on a class and select Insert code..., Add JavaFX property...



TableView with images

- Injected fields

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

- In the initialization of the controller:

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

TableView with images

- For the city column, also in the initialize method:

```
// What information is shown?  
cityColumn.setCellValueFactory(c -> c.getValue().residenceProperty());  
  
// How is the information displayed?: use a CellFactory  
cityColumn.setCellFactory(v -> 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().toUpperCase());  
        }  
    }  
});
```

We show the city name
in uppercase

Declared as its column

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

TableView with images


- For the column with the image:

```
// What information is shown?
```

```
imageColumn.setCellValueFactory(c -> c.getValue().pathImageProperty());
```

```
// How is the information displayed?
```

```
imageColumn.setCellFactory(c -> 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);  
        }  
    }  
});
```



Load the image file.
item contains its path

- Former code Works if the image is located in folder resources of the Project, otherwise see next slide

TableView with images

- If the image is located outside the jar of the project

```
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 with attributes

- Suppose domain class Person contains both properties and non-property data fields

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

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

- Injected fields now are

```
@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 with attributes

- Code for the visualization

```
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()));
```

- Note that for properties, the expression below does not generate exceptions in case there is no property with the given name. Therefore it is preferable to use the form above

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

Exercise

- Change the interface in the project of the ListView of Persons, for displaying the list of persons in a TableView.
- Create the list of persons in the main class and pass it to the controller
- Include in the interface the following buttons: Add, Modify and Remove
 - The add and modify actions should be performed in a popup window

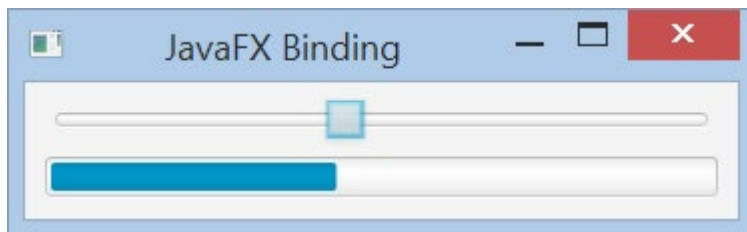
Exercise (and 2)

- If you have time, modify the exercise for showing an icon for each person
- You can download a ZIP with 3 images in poliformaT, or use your own
- Insert the images in a folder of your project (for example, /images/). The Person class should have the full pathname for an image:

```
new Person(1000,  
    "John Doe",  
    new Residence("Museros", "Valencia"),  
    "/images/Lloroso.png")
```

ANEX I. Binding of properties

- A binding synchronizes the values of the properties:
 - If a property A is bound with a property B, any change in B is reflected in A ($A=f(B)$)
- **One way** binding is done by `bind()`
- **Two ways** binding is done by `bindBidirectional()`
- For removing bindings: `unbind()` and `unbindBidirectional()`
- Example: Binding the `progressProperty` of a `ProgressBar` with the `valueProperty` of a `Slider`



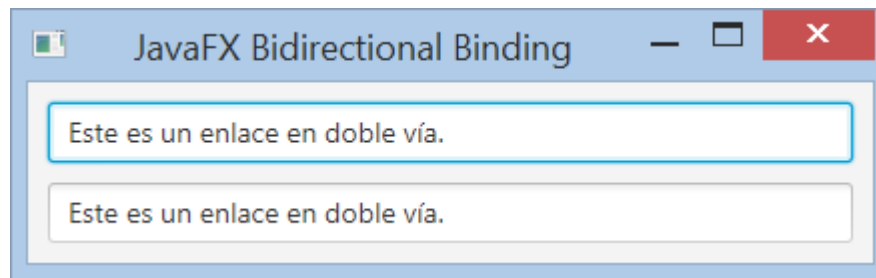
While the `progressProperty` is bound, any change by code in the value of the property produces an error

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

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

Binding of properties

- Another example, adding a **bidirectional** binding between the content of two text fields:



```
@FXML TextField tf_1;  
@FXML TextField tf_2;
```

```
// in the controller initialization method
```

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

- All the changes in one edit field are reflected in the other one

Numeric Binding of properties

- Numerical properties can be linked:

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

```
NumberBinding sum = x.add(y);  
int valor = sum.intValue();  
// sum = x+y compiling error!!!
```

- It is possible to access to the sum value using: `intValue()`, `longValue()`, `floatValue()`, `doubleValue()` for getting the values as int, long, float or double.
- An equivalent way:

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

Numeric Binding of properties

- The radius of a circle is linked to the height and width of the gridPane in which is located:

```
CirculoFXID.radiusProperty().bind(  
    Bindings.min(gridPaneFXID.widthProperty(),  
        gridPaneFXID.heightProperty()).divide(5).divide(2));
```

Utility class

API Fluent, allows us to concatenate operations

```
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));
```

References

ListView and TableView Oracle

<http://docs.oracle.com/javase/8/javafx/user-interface-tutorial/list-view.htm>

<http://docs.oracle.com/javase/8/javafx/user-interface-tutorial/table-view.htm>

JavaFX UI Controls

http://docs.oracle.com/javase/8/javafx/user-interface-tutorial/ui_controls.htm