



## P4. MODELS AND DATA VIEWS

---

Interfaces Persona Computador

Depto. Sistemas Informáticos y Computación

UPV

# Outline

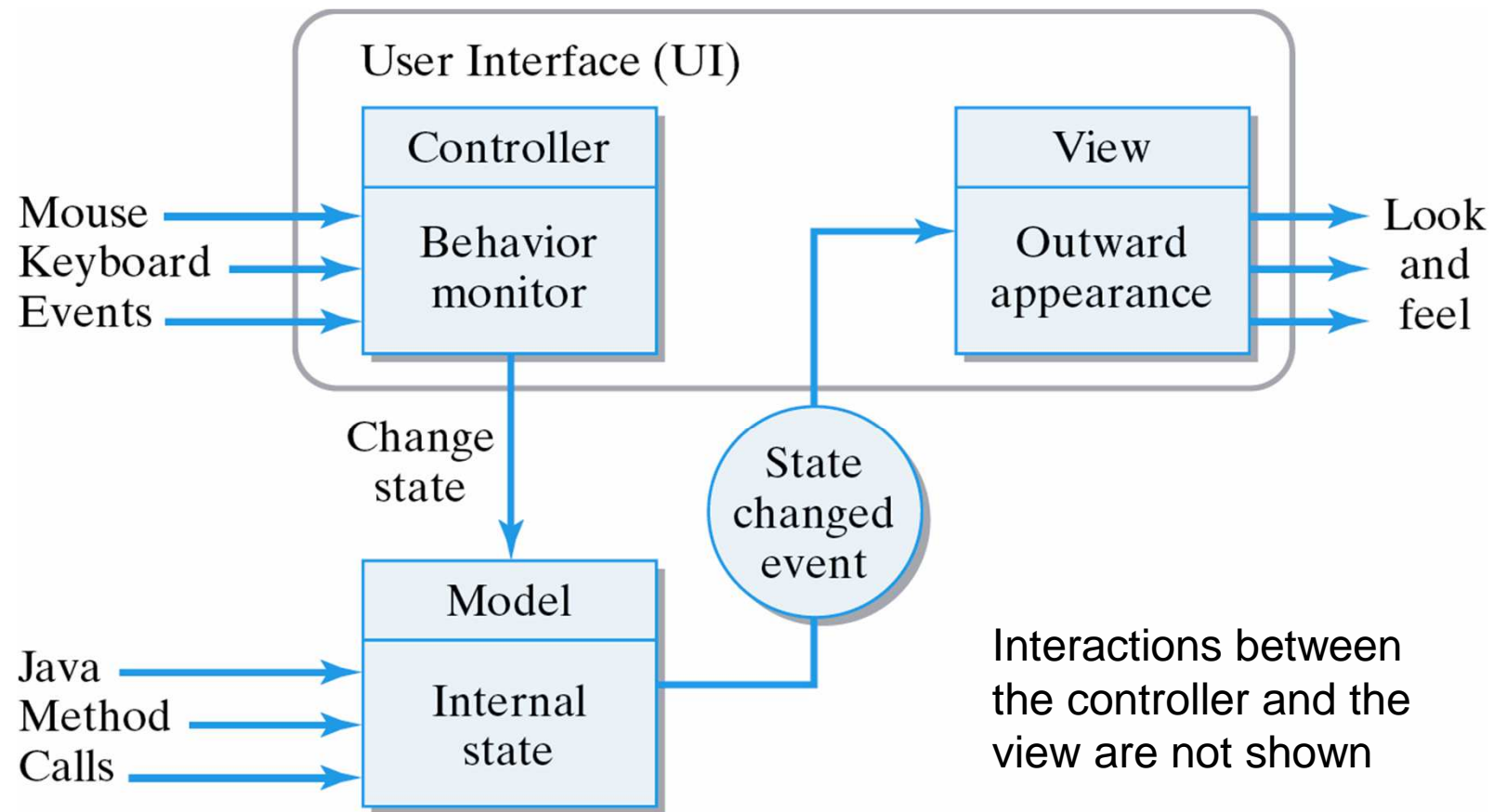
- Introduction
- Collections in JavaFX
  - ListView
  - ListView with images
- Passing parameters to a controller
- Applications with multiple windows
  - One stage and several scenes
  - Several stages with their scenes
- Exercise
- Additional graphical components
  - TableView
  - TableView with images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

# 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

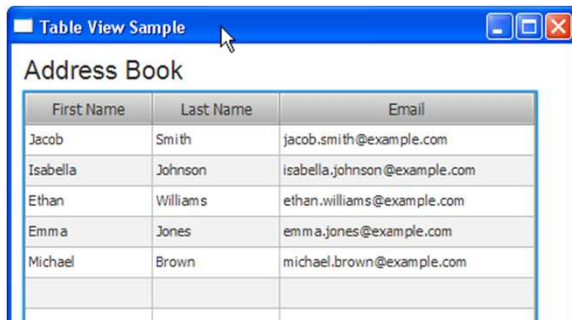
- Relationships



# 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



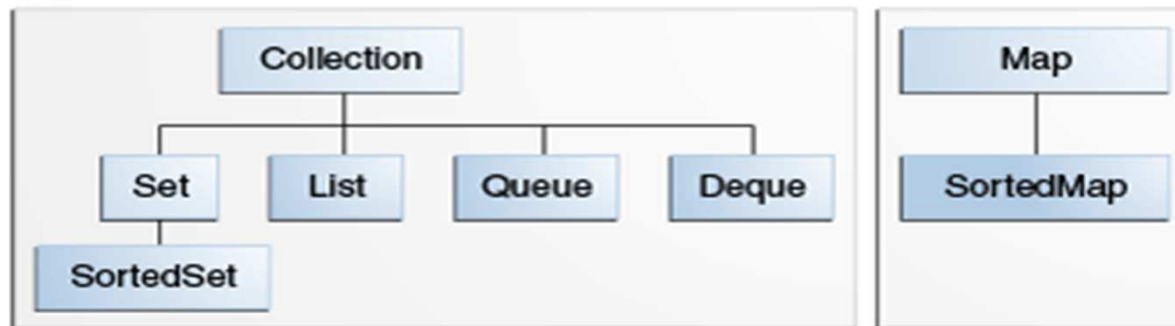
First Name	Last Name	Email
Jacob	Smith	jacob.smith@example.com
Isabella	Johnson	isabella.johnson@example.com
Ethan	Williams	ethan.williams@example.com
Emma	Jones	emma.jones@example.com
Michael	Brown	michael.brown@example.com

## 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:



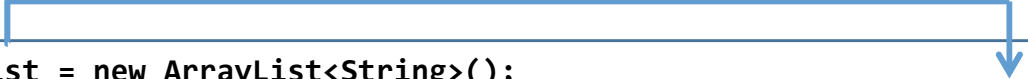
Interface	Hash	Array	Tree	Linked list	Hash+ Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

# 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 = FXCollections.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());
```


```
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
```

# 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

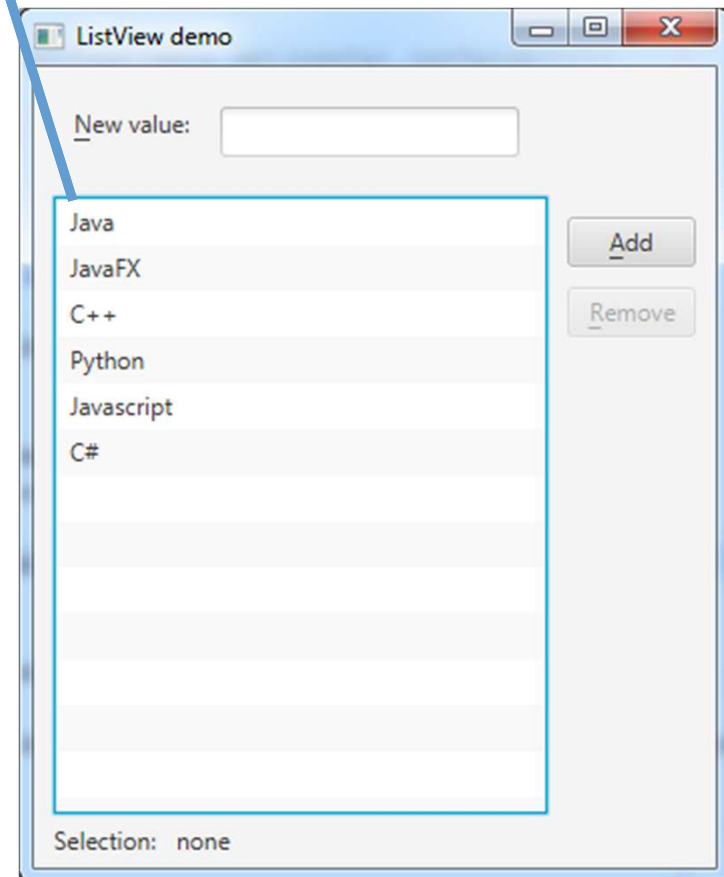
```
private ObservableList<String> data = null;  
...  
data = FXCollections.observableArrayList(myData);
```

Bind to listview

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

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

ListView



# Example of ListView

- 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

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

# Example of ListView

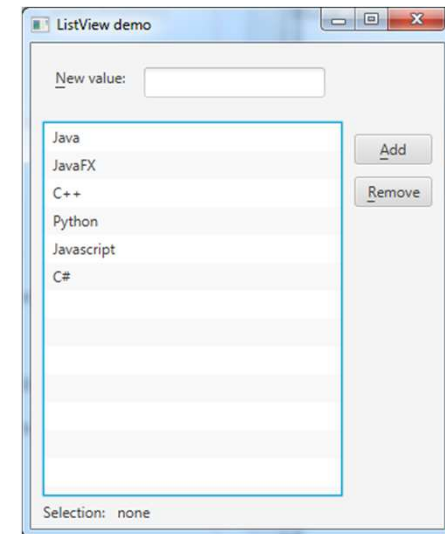
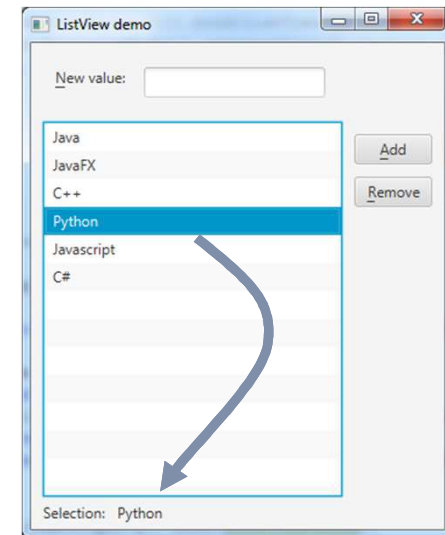
- Listening to changes in selection

- Option 1

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

- Option 2

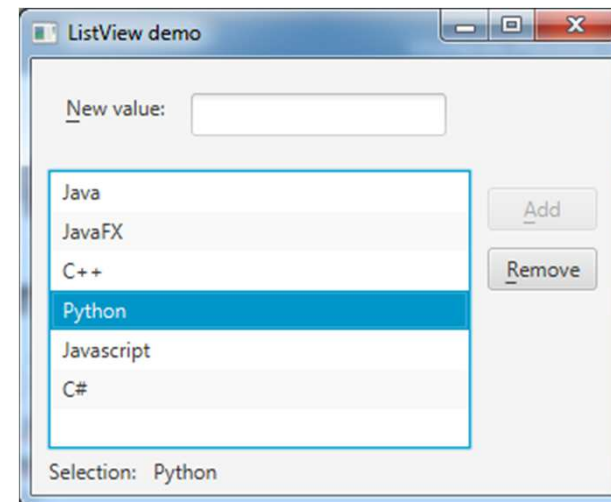
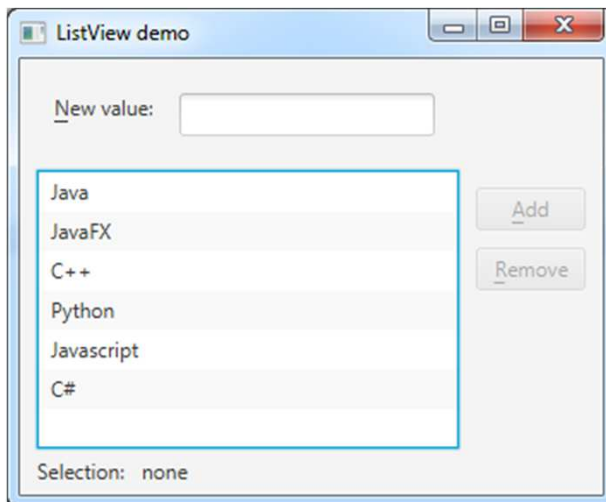
```
listView.getSelectionModel().selectedIndexProperty().  
    addListener( (o, oldVal, newVal) -> {  
        if (newVal.intValue() == -1)  
            selectedItem.setText("none");  
        else selectedItem.setText(  
            listView.getItems().get(newVal.intValue()));  
    });  
selectedItem.setText("none");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



# 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(inputText.textProperty().isEmpty());
```

- Buttons can also be manually disabled/enabled with:

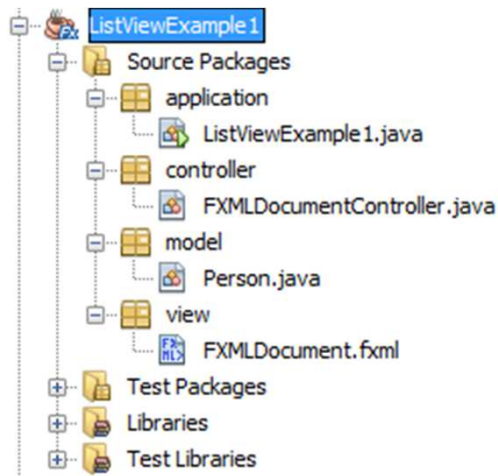
```
addButton.setDisable(true);
```

```
removeButon.setDisable(false);
```



# Example of ListView

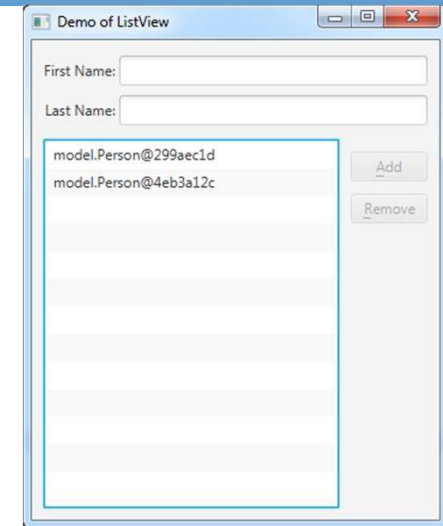
- Download the example from poliformaT (`ListViewExample1.zip`) and import it into NetBeans. The project will have the following structure:



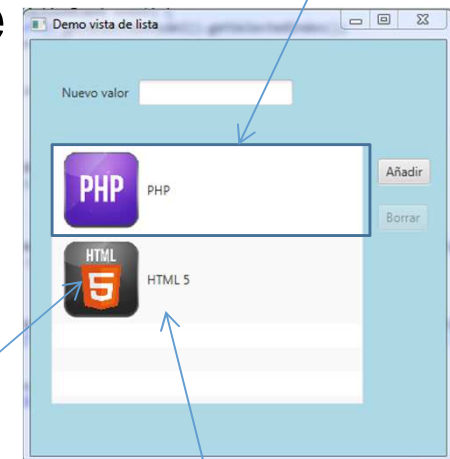
- Note the organization in different packages

# Example of ListView

- The result of running the program is:
- ListView by default shows strings (if we ask it to show an object, it will use the `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



Cell



Image

Text

# ListView: Cell and CellFactory

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

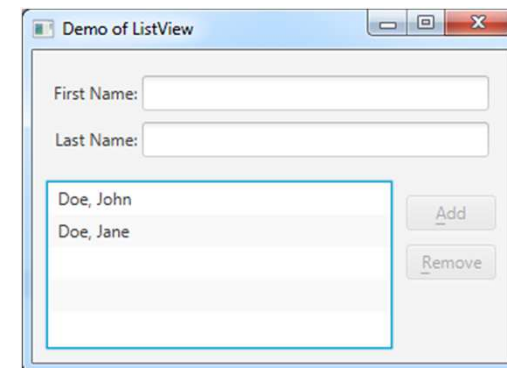
The class in charge of showing a cell

```
// 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

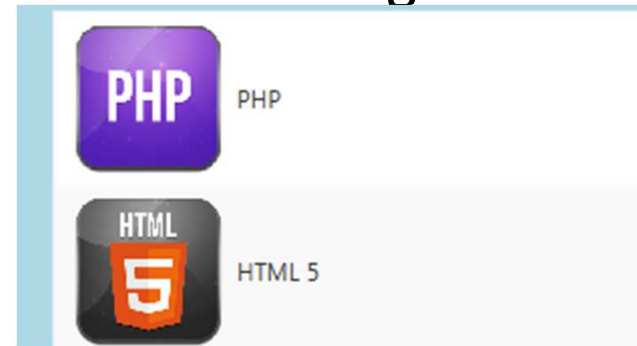
```
listView.setCellFactory(c-> new PersonListCell());
```



# 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;
```

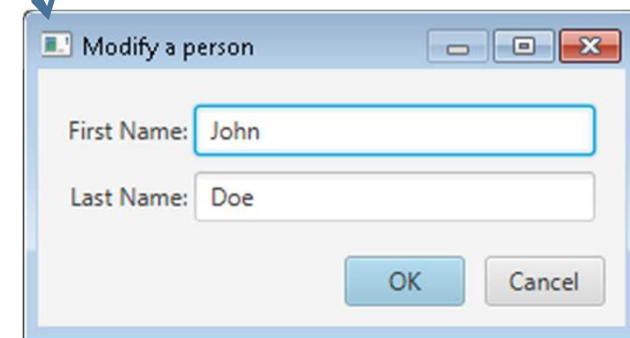
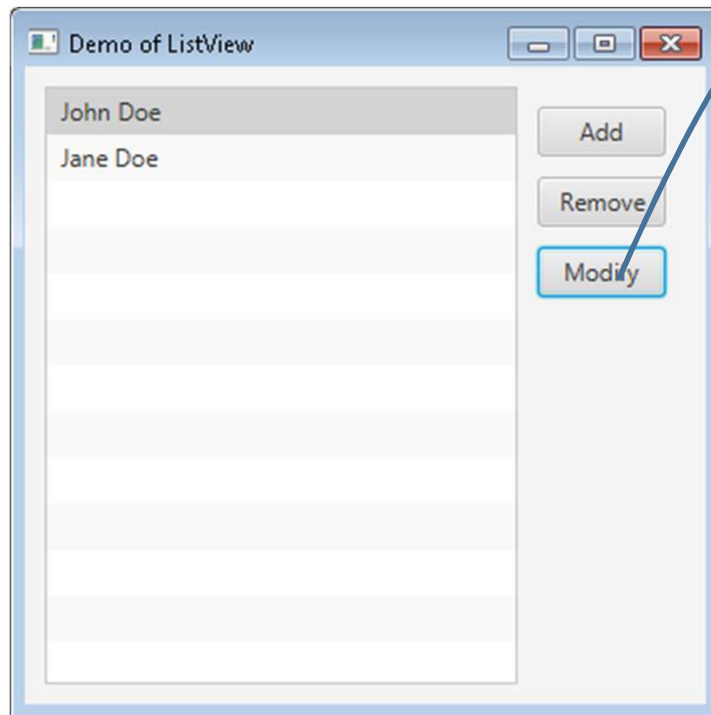
```
...
```

- 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();
```

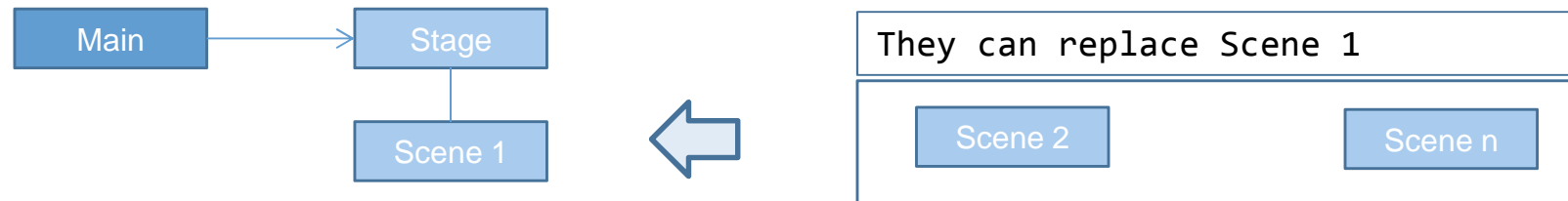
```
// obtain a reference to the controller  
PersonDataController personController = myLoader.<PersonDataController>getController();  
personController.initPerson(person);
```

```
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. It could be in the Main class if we pass data from there

# 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

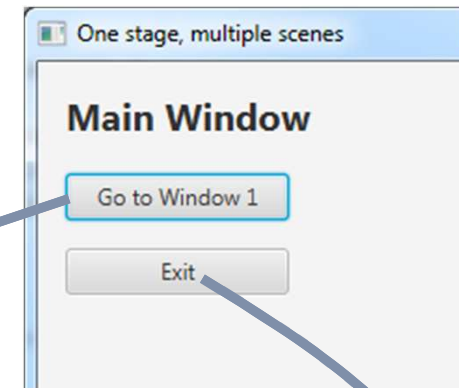
```
public class SingleStage extends Application {  
    @Override  
    public void start(Stage stage) throws Exception {  
        FXMLLoader loader =  
            new FXMLLoader(getClass().getResource("/view/MainWindow.fxml"));  
        Parent root = loader.load();  
        Scene scene = new Scene(root);  
        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

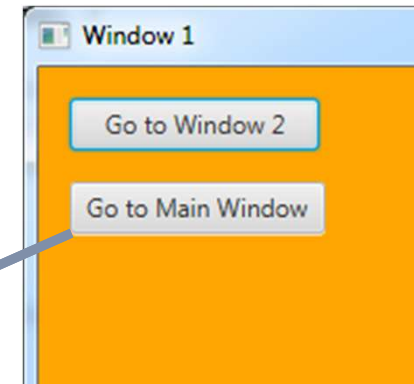
```
public class MainWindowController implements Initializable {
    private Stage primaryStage;
    public void initStage(Stage stage) {
        primaryStage = stage;
    }
    @FXML
    private void onGoToWindow1(ActionEvent event) {
        try {
            FXMLLoader myLoader = new FXMLLoader(getClass().getResource("/view/Window1.fxml"));
            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

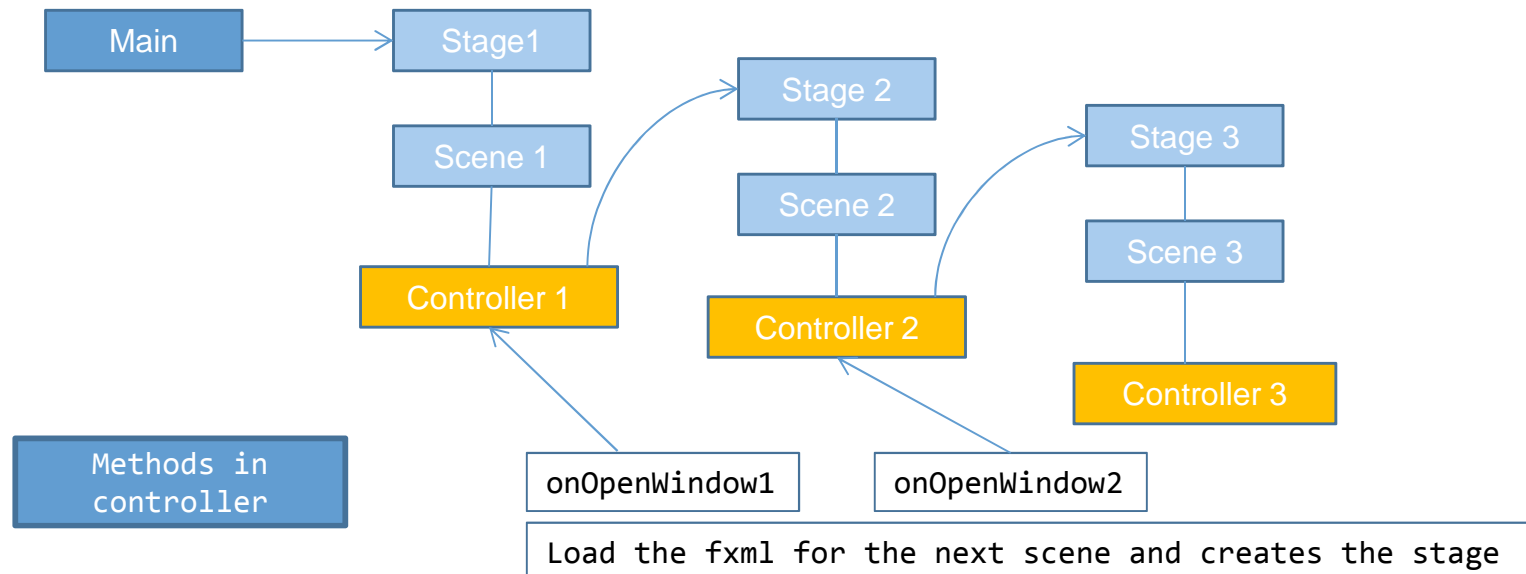
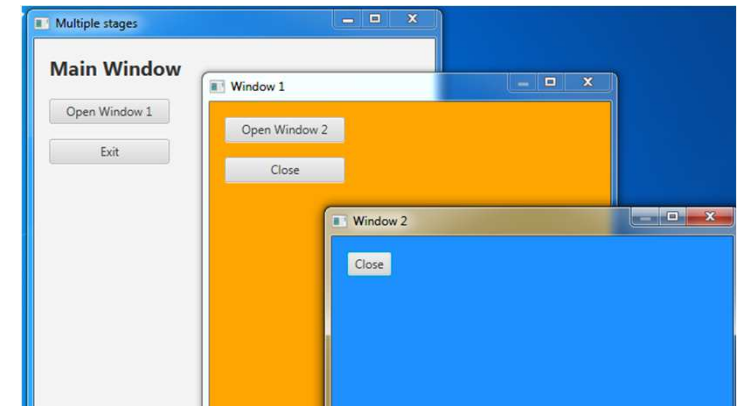
- Controller class for Window 1:

```
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

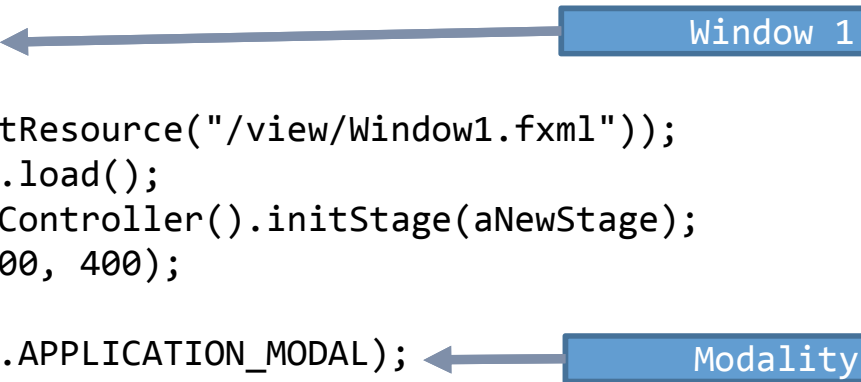
- The code of the main class is similar to the previous example
- Each scene has its own stage

```
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();
    }
}
```



# Applications with multiple windows

- Controller class for Window 1

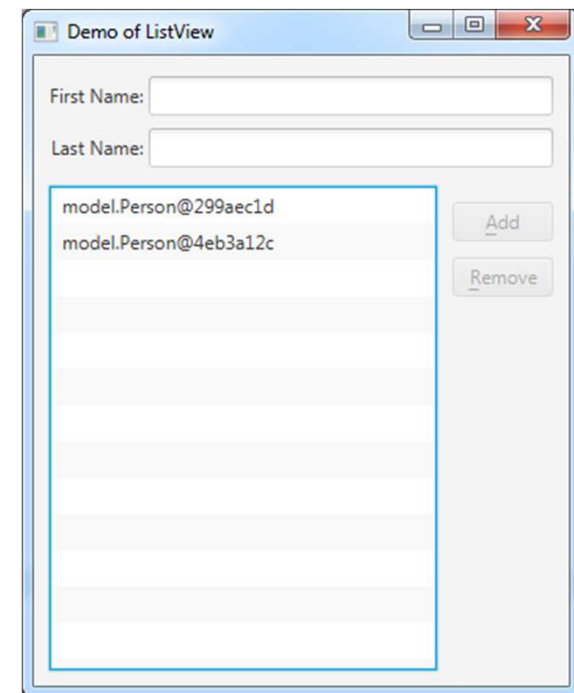
```
public class Window1Controller implements Initializable {
    private Stage myOwnStage;
    public void initStage(Stage stage) {
        myOwnStage = stage;
        myOwnStage.setTitle("Window 1");
    }
    @FXML private void onOpenWindow2(ActionEvent event) {
        try {
            Stage aNewStage = 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 arrow points from the line `Stage aNewStage = new Stage();` in the `onOpenWindow2` method to a blue rectangular box labeled "Window 2". This indicates that the `new Stage()` call creates a new window instance.

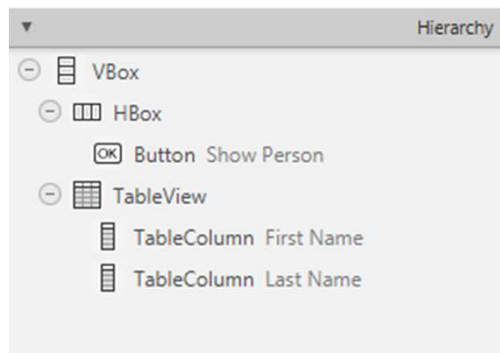
# Exercise

- Start with the ListView example that uses the Person class (ListViewExample1.zip):
  - Make the list to show for each person:  
<First name> <Last name>
  - Create a new view with the fields First and Last name. Remove those fields from the original window.
  - Make the Add button to be enabled at all times
  - Add a Modify button
  - 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



# TableView

- The control shows rows of data, where each row is divided into columns
- **TableColumn** represent a column in the table and contains a **CellValueFactory** for defining how the cell will be shown
- If the table only contains text columns:
  - Scene Builder



Assigned fx:id	
fx:id	Component
tableView	TableView
firstNameColumn	TableColumn
lastNameColumn	TableColumn

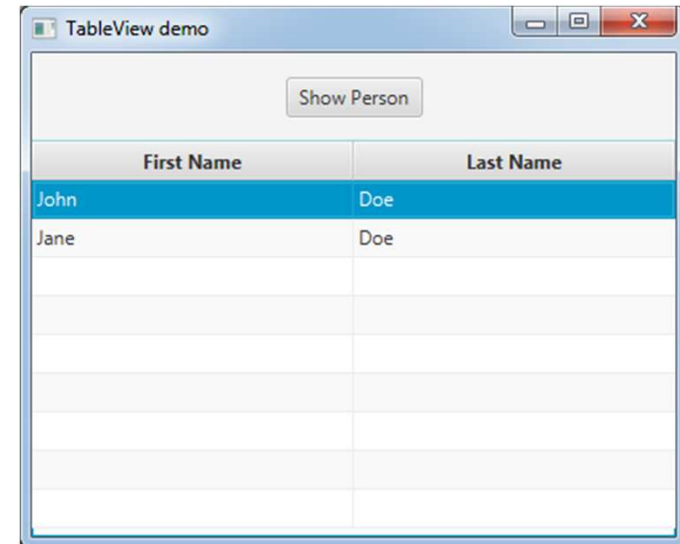
3 items





# 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 will be shown  
as 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);
```

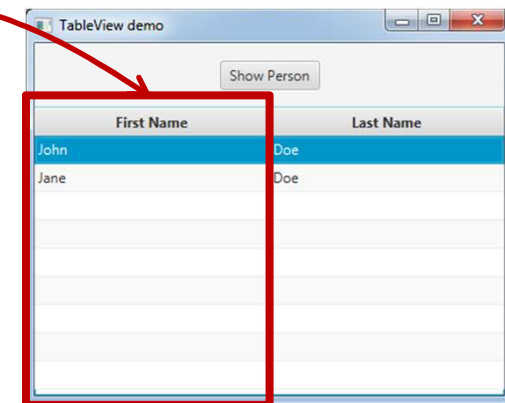
- The `PropertyValueFactory<Person,String>(String prop)` CLASS:
  - Is a convenience class for extracting a property from a `Person`
  - Internally, it will try to invoke: `<prop>Property()`, `get<prop>` or `is<prop>` in the `Person` object to be shown

# 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 = ...  
    private final StringProperty lastName = ...  
  
    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

- 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();  
        }  
    });
```

- Or:

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

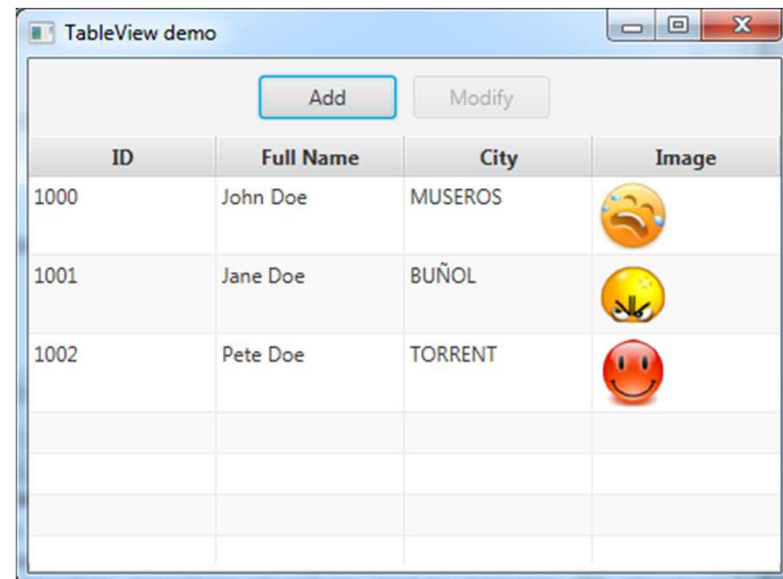
# TableView with images

- 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

# 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")
```

# XML in JavaFX (I)

Why to use XML instead of databases?

- They are one of the most used methods to store information
- Databases typically use the relational model (tables linked by indices) to organize data
- But this model may be inadequate if data is structured as object (object-relational impedance mismatch).

# XML in JavaFX (II)

## Why to use XML?

- It is easier for our simple data model
- Library [JAXB](#) (*Java Architecture for XML Binding*).
- JAXB allows to create the following output with a little bit of code:

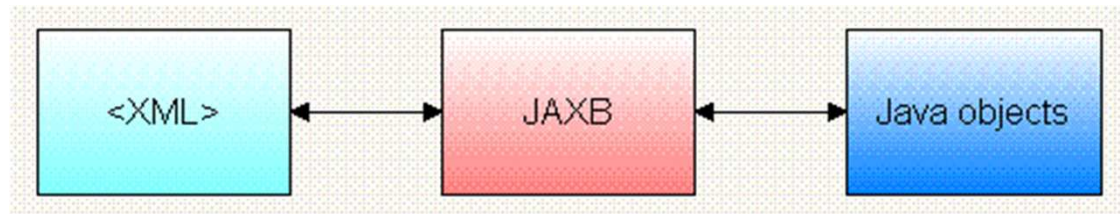
```
<persons>
  <person>
    <birthday>1999-02-21</birthday>
    <city>some city</city>
    <firstName>Hans</firstName>
    <lastName>Muster</lastName>
    <postalCode>1234</postalCode>
    <street>some street</street>
  </person>
</persons>
```

```
<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>
```

# XML in JavaFX (III)

## JAXB

- Included in the JDK. No external library is required
- Provides two main functions:
  - **Marshalling**: converting Java objects to XML.
  - **Unmarshalling**: converting XML to Java objects.



- The model of our application has to be prepared for helping JAXB do its work, adding annotations, for example, the root class (the one that contains the whole XML tree) must be annotated with **@XmlElement**.

# XML in JavaFX

- The required annotations are:

Annotation	Meaning
<code>@XmlAccessorType(PUBLIC_MEMBER, PROPERTY, FIELD, o NONE)</code>	Access type for binding the XML with the properties and fields of the class
<code>@XmlRootElement(namespace = "namespace")</code>	Defines the XML file root
<code>@XmlType(propOrder = { "field2", "field1",.. })</code>	Indicates the order in which the class attributes will be saved
<code>@XmlElement (name = "name")</code>	Indicates that the attribute will be saved
<code>@XmlAttribute (name = "name")</code>	Specifies an attribute for the XML root element
<code>@XmlTransient</code>	The attribute won't be saved

# Examples in JAXB

- Given the following class

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

- We should follow these steps to save an instance from Person to XML:
  - Add a constructor without parameters to Person
  - Annotate the class with `@XmlElement`
  - The class Residence, does not need annotations because it is not the root of the XML file



# Examples in JAXB

- The result is:

`@XmlElement`

`public class Person {`

`private final StringProperty fullName = new SimpleStringProperty();`

`private final IntegerProperty id = new SimpleIntegerProperty();`

`private List<Residence> residences;`

`private final StringProperty pathImage = new SimpleStringProperty();`

`public Person() { }`

`... // All the getters and setters of the properties`

The only required  
annotation

Information stored in the XML

- When there is no `@XmlAccessorType` only the public fields and the public members (pairs of getter/setter) are saved
- The name of the root element is the class name by default. It can be changed with:

```
@XmlElement ( name = "person" )
```

# Examples in JAXB

- How do we save an instance of Person?

```
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");

// Save to disk
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();
}
```

# Examples in JAXB

- Contents of the file

```
<?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>
```

# Examples in JAXB

- The XML file can be read to recreate the instance of a person

```
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();  
}
```

# Examples in JAXB

- What if we want to store a list of persons in an XML file?
- JavaFX's FXCollections cannot be mapped directly into XML
- The list must be wrapped inside of a container class:

```
@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;
    }
}
```

# Examples in JAXB

- The code for saving the wrapper object is similar to the previous example

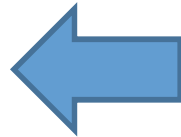
```
ListPersonWrapper listToSave = new ListPersonWrapper();
listToSave.setPersonList(theList);

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();
}
```

# Examples in JAXB

- The result XML file contains:

```
<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>
```



```
<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>
```

# References

## **ListView Oracle**

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

## **JavaFX UI Controls**

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

## **XML in Wikipedia**

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

## **Introduction to XML in w3schools**

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