CMPS 251

Read Chapter 12 and 13

Graphical User Interfaces (GUI)

Dr. Abdelkarim Erradi

CSE@QU

Outline

- Commonly used JavaFX UI Components
- Properties and Bindings
- Multi-scenes / Multi-windows App

Commonly used JavaFX UI Components

- _6.controls.combobox
- _6.controls.listview
- _6.controls.piechart
- _6.controls.radiobutton
- _6.controls.registrationform
- _6.controls.tableview

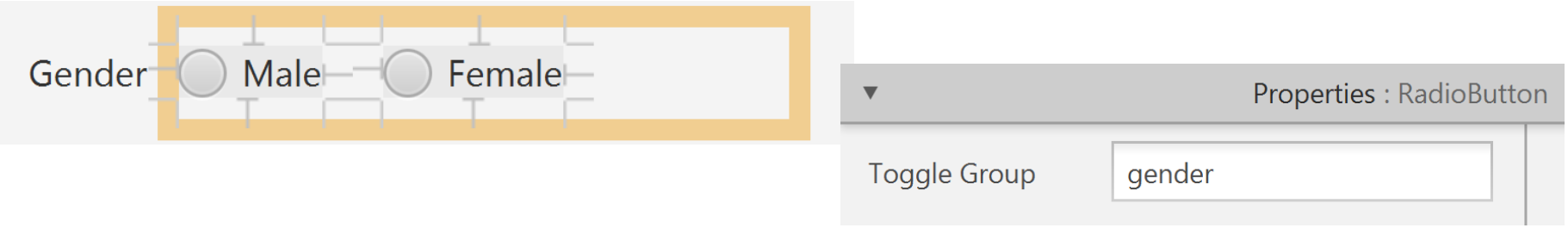
Back

Commonly used JavaFX UI Components

- Label, Button, RadioButton, ToggleButton
- CheckBox, ChoiceBox
- TextField, PasswordField, TextArea, Hyperlink
- ListView, TableView
- MenuBar, MenuButton, ContextMenu, ToolBar
- ImageView, Audio Player, Video Player
- … see posted examples …

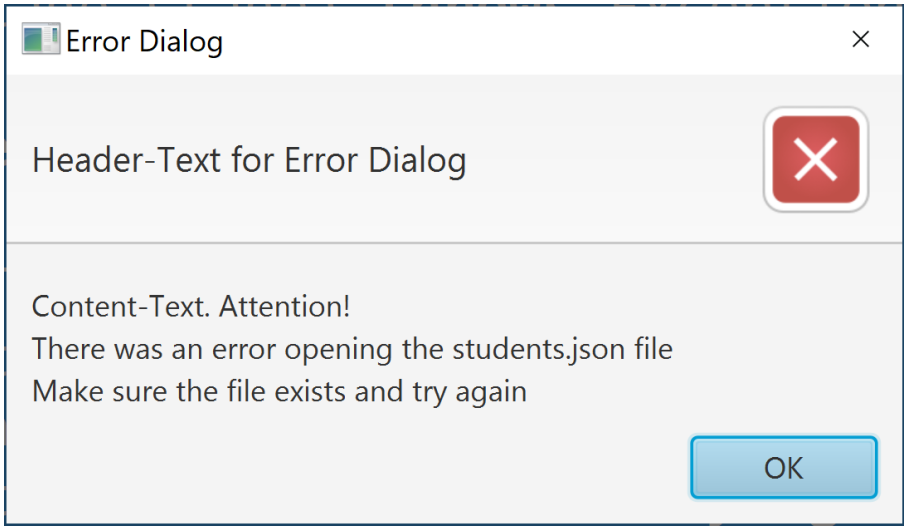
Radio Button

- To group radio button and allow the user the make mutually exclusive choice, select the radio buttons to group and assign them the same '**Toggle Group**' name



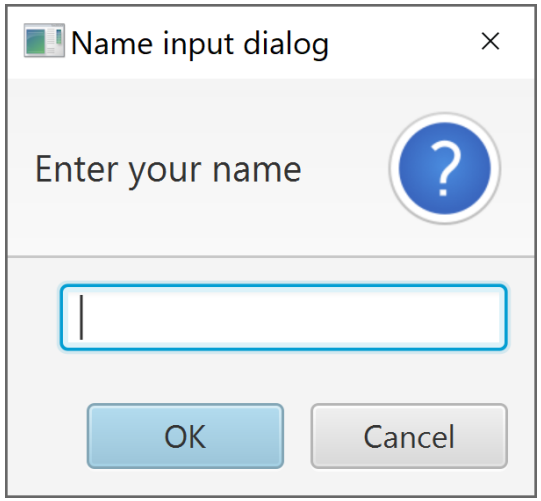
Info/Warn/Error Dialog

```
public void start(Stage stage) throws Exception
{
 Alert alert = new Alert(AlertType.ERROR);
 alert.setTitle("Error Dialog");
 alert.setHeaderText("Header-Text for Error Dialog");
 alert.setContentText("Content-Text. Attention!\n" +
  "There was an error opening the students.json   file\n" +
   "Make sure the file exists and try again");
 alert.showAndWait();
}
```



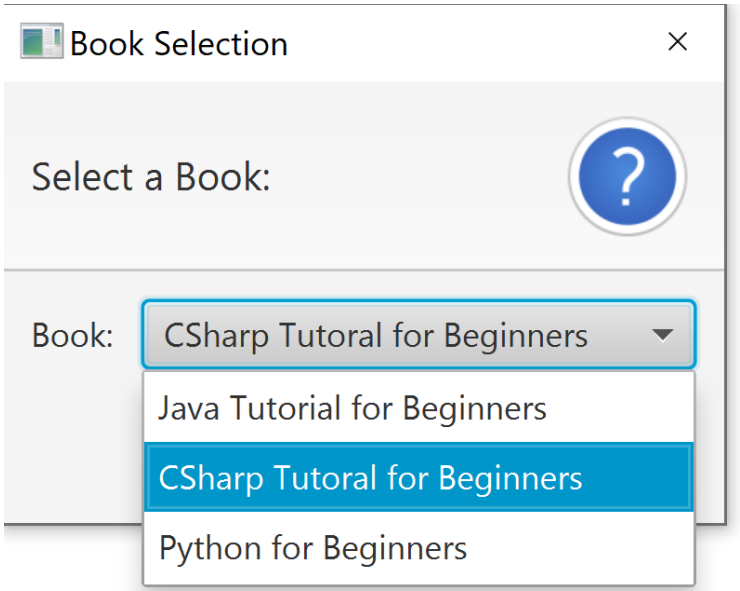
Input Dialog

```
public void start(Stage stage) throws Exception
{
 TextInputDialog dialog = new TextInputDialog();
 dialog.setTitle("Name input dialog");
 dialog.setHeaderText("Enter your name");
 Optional<String> result = dialog.showAndWait();
 result.ifPresent(name ->
      System.out.println("Your name: " + name));
}
```



Choice Dialog

```
List<Book> books = List.of(java, csharp, python);
Book defaultBook = csharp;
ChoiceDialog<Book> dialog = new ChoiceDialog<Book>(defaultBook, books);
dialog.setTitle("Book Selection");
dialog.setHeaderText("Select a Book:");
dialog.setContentText("Book:");
Optional<Book> result = dialog.showAndWait();
result.ifPresent(book -> System.out.println(book.getName()) );
```

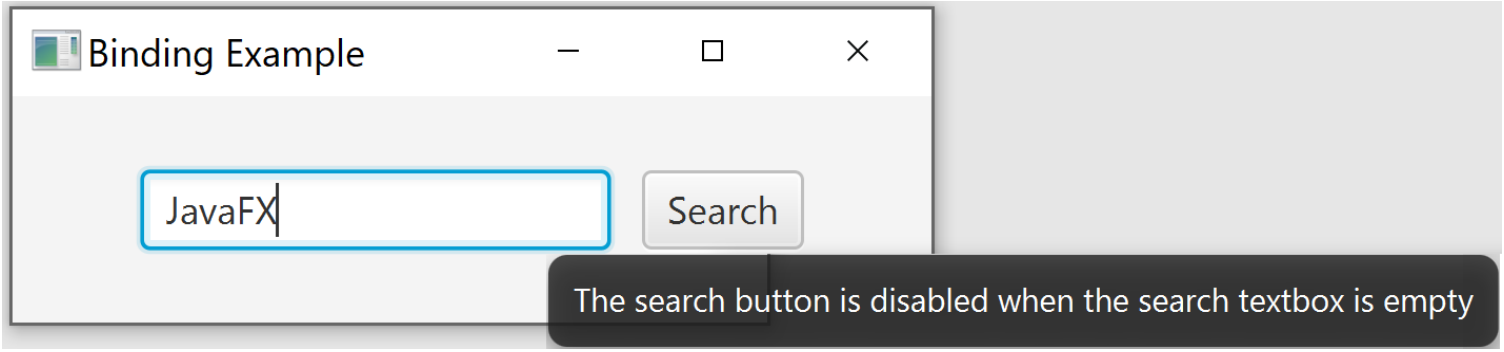


Tool Tips

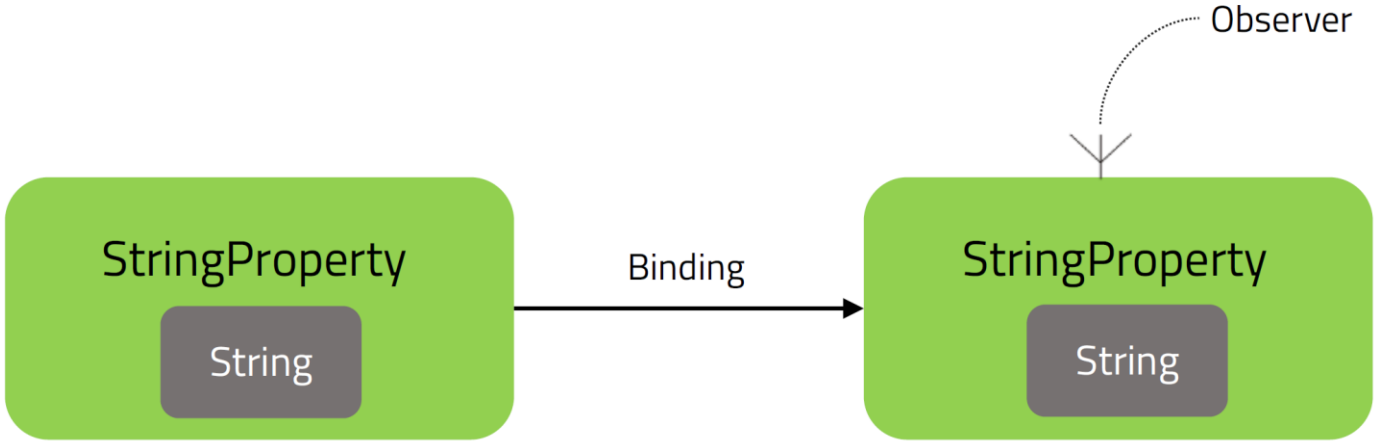
- A *tool tip* provides a short pop-up description when the mouse cursor rests momentarily on a component

- A tool tip is assigned using the `setTooltip` method of a JavaFX control

```
Tooltip tip = new Tooltip("The search button is
        disabled when the search textbox is empty");
searchButton.setTooltip(tip);
```



Properties and Bindings



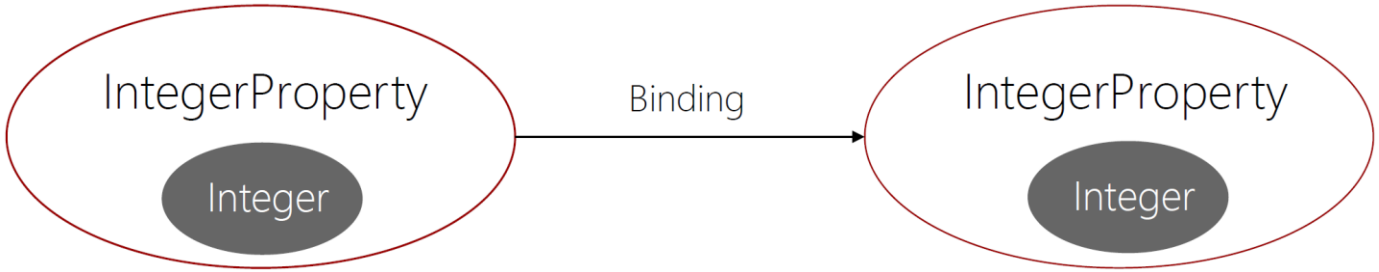
Excel is an excellent example …

Properties

- A JavaFX *property* is an object that holds a value
- A property is ***observable:*** when a property's value changes, other objects can respond accordingly
- So, instead of holding an `int` value in integer primitive type, make it a **property** and store in `IntegerProperty` object
- Most values of JavaFX classes, such as the width of a `TextField`, are stored as properties
- A key benefit of properties is ***property binding***

Property Binding

- Property binding enables propagating changes
  - The target listens for changes in the source and updates itself when the source changes
  - Binding syntax: `target.bind(source);`



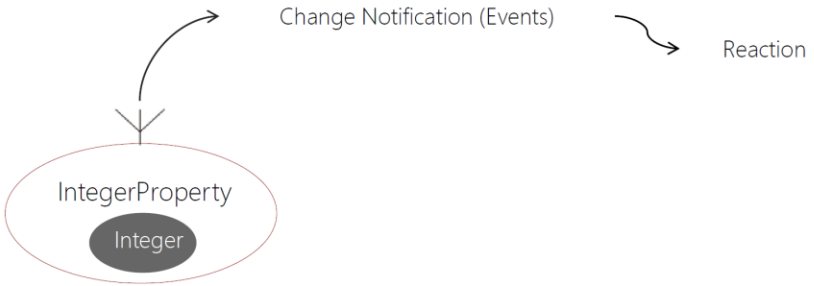

```
IntegerProperty num1 = new SimpleIntegerProperty(1);
IntegerProperty num2 = new SimpleIntegerProperty(2);
num1.bind(num2);
System.out.println("num1 is " + num1.getValue()
             + " and num2 is " + num2.getValue());

num2.setValue(15); //Changing num2 will auto-update num1

System.out.println("num1 is " + num1.getValue()
             + " and num2 is " + num2.getValue());
```

Property Binding

- A pair of simple integer property objects (not int values) are created with different values. Then one is **bound** to the other
  - If the value of one is changed then the other will also be changed.
- Can also listen to value change events

```
intProp.addListener((prop, oldVal, newVal) ->
        System.out.println(oldVal+ " "+ newVal));
```

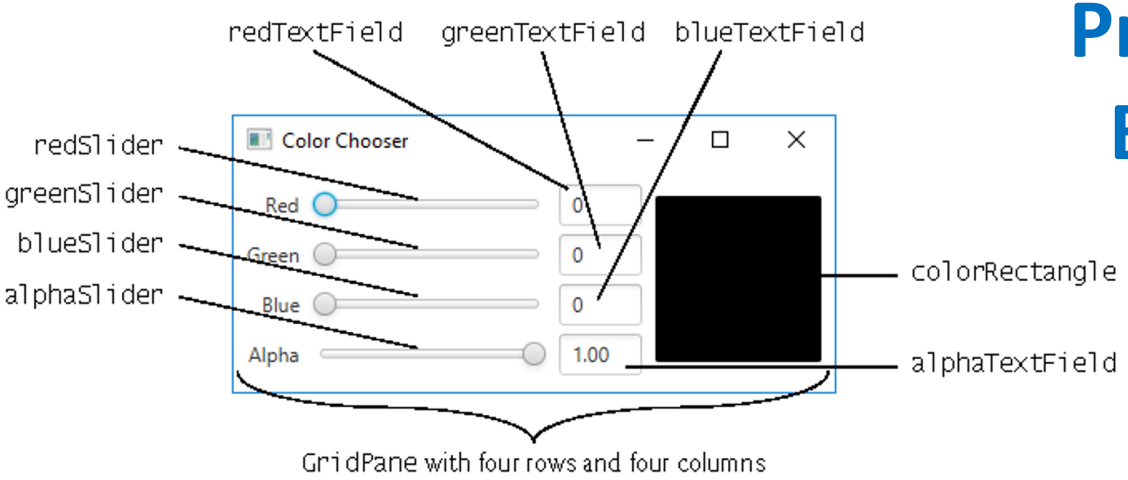
- Property Binding is used to **synchronize** the UI and the associated objects



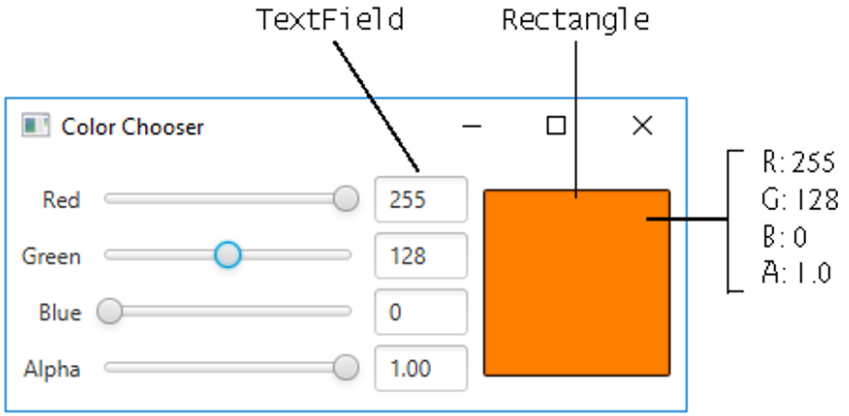
Unidirectional vs. Bidirectional Binding

- Property-to-Property Bindings can be Unidirectional vs. Bidirectional
- For example the `text field` value and `slider` value **binded bidirectionaly** which means if text field value changes or slider value changes it will affect another one.
- In the other hand, `progress indicator field` and `slider` value **binded unidirectionaly** means that only changes in slider value will affect Progress Indicator.

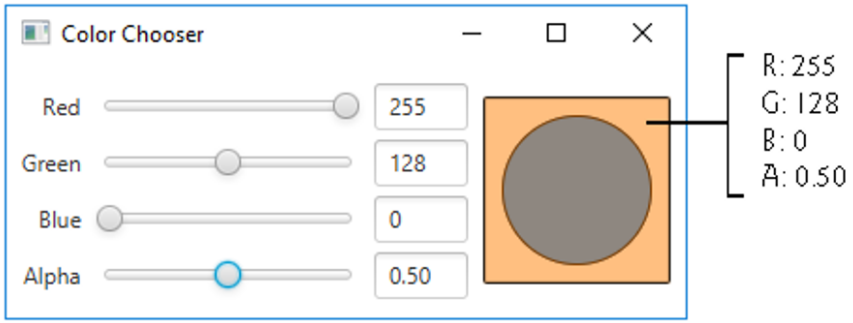
Property-to-Property Bindings - Example



a) Using the **Red** and **Green** Sliders to create an opaque orange color



b) Using the **Red**, **Green** and **Alpha** Sliders to create a semitransparent orange color—notice that the semitransparent orange mixes with the color of the circle behind the colored square



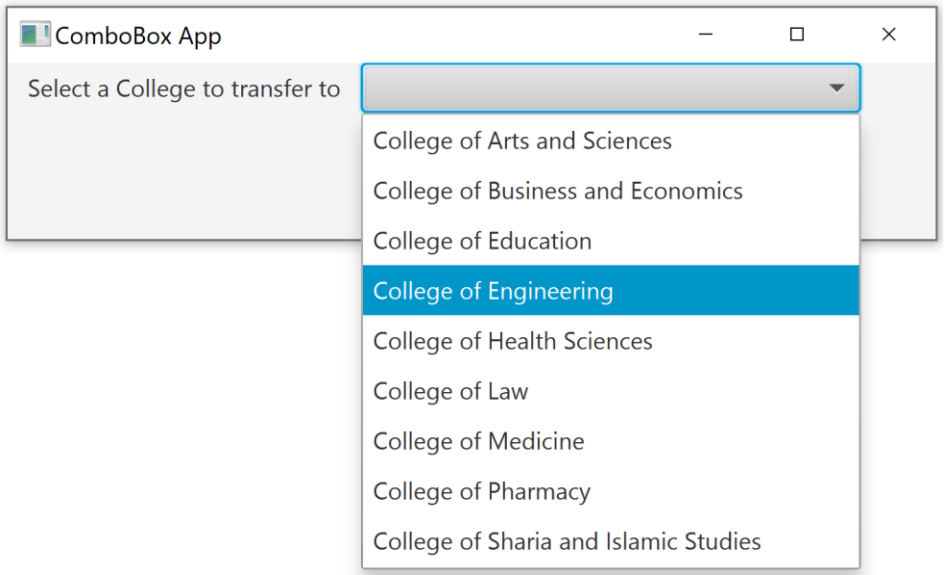
Binding to Class Properties

```
public class Conference {
 private StringProperty name = new SimpleStringProperty();
 public StringProperty nameProperty() { return this.name; }
 public String getName() { return this.name.get( ); }
 public void setName(String name) { this.name.set(name); }
}

final Conference conf = new Conference();
nameTf.textProperty().bindBidirectional(conf.nameProperty());
```

Fill a ComboBox using an ObservableList

```
@FXML private ComboBox<String> collegesCombo;

public void initialize() {
 ObservableList<String> colleges =
  FXCollections.observableArrayList(CollegeRespository.getColleges());
 collegesCombo.setItems(colleges);
}
```

Observable = **notifies** the UI when the underlying list changes

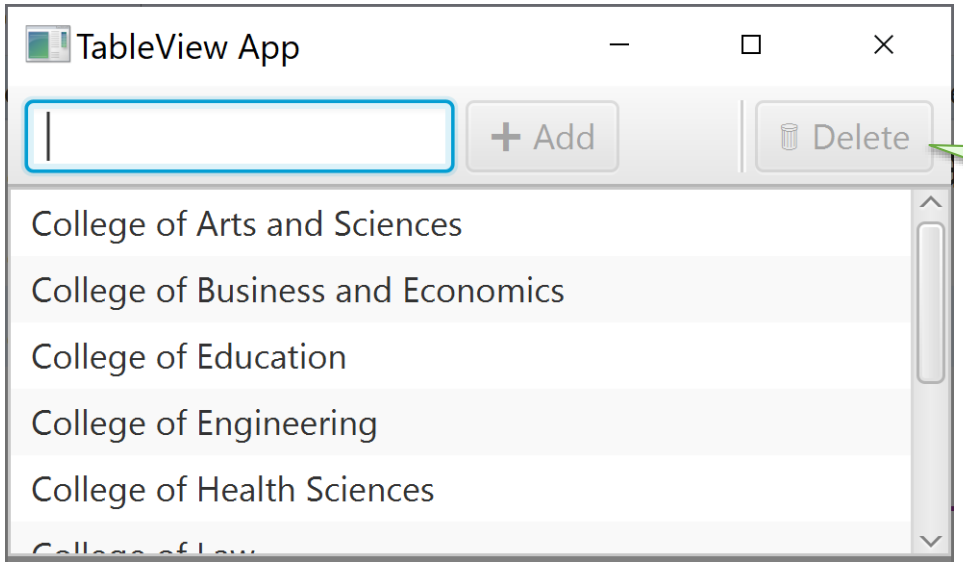


ObservableList

- The previous example uses a ComboBox control to display a list of names
- To fill a ComboBox you can pass an ObservableList object to the ComboBox using **`setItems`** methods
  - If you make changes to an ObservableList, its **observer** (the ComboBox in this app) will automatically be notified of those changes
- Package javafx.collections contains **FXCollections** class, which provides static methods for creating and manipulating observable collections

Fill a ListView using an ObservableList

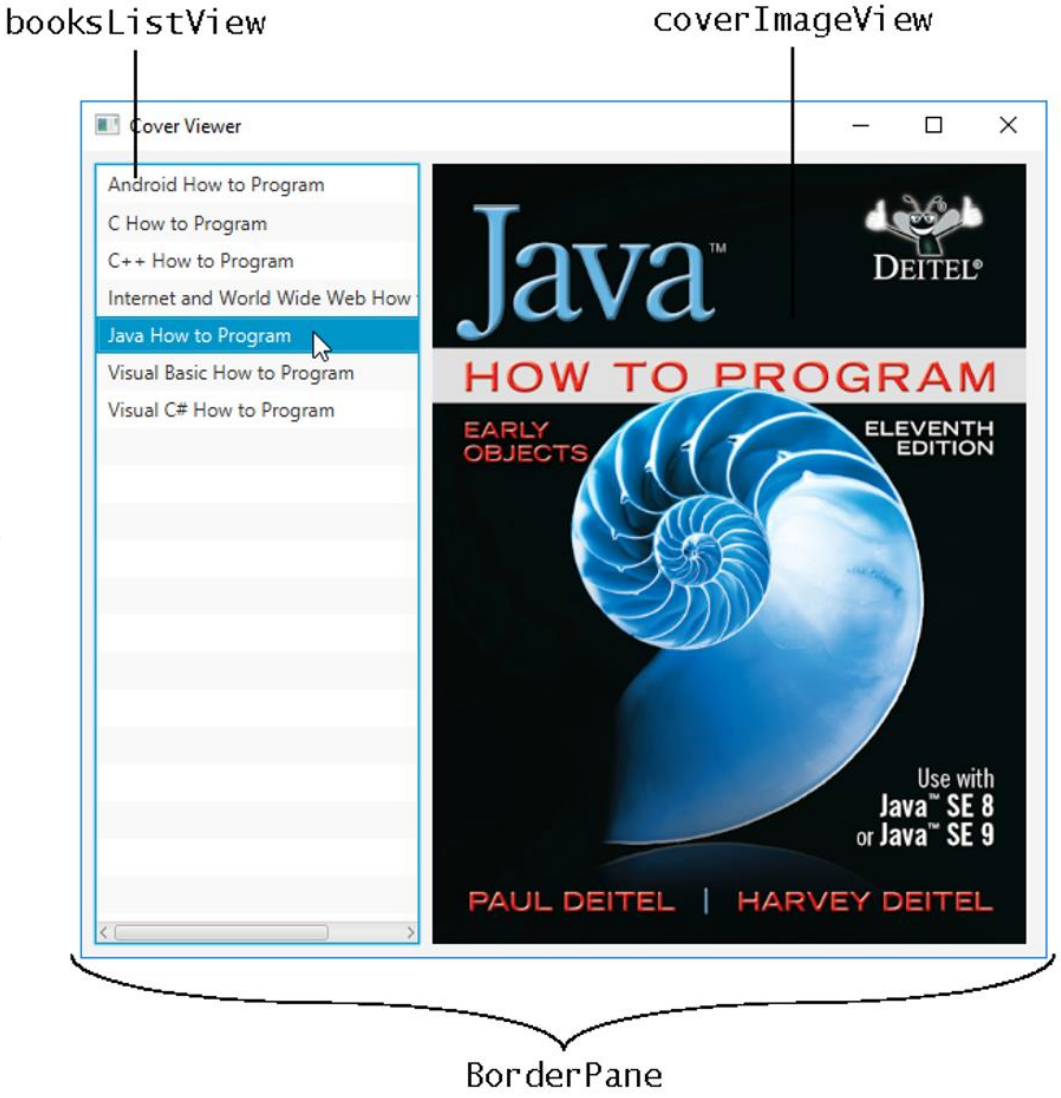
```
@FXML private ListView<String> collegesList;
@FXML private Button deleteButton;
public void initialize() {
  ObservableList<String> colleges =
        FXCollections.observableArrayList(CollegeRespository.getColleges());
  collegesList.setItems(colleges);
  //If no student selected then disable to delete button
  deleteButton.disableProperty().bind( Bindings.isNull(
        collegesList.getSelectionModel().selectedItemProperty()) );
}
```



When no name is selected the delete button is disabled

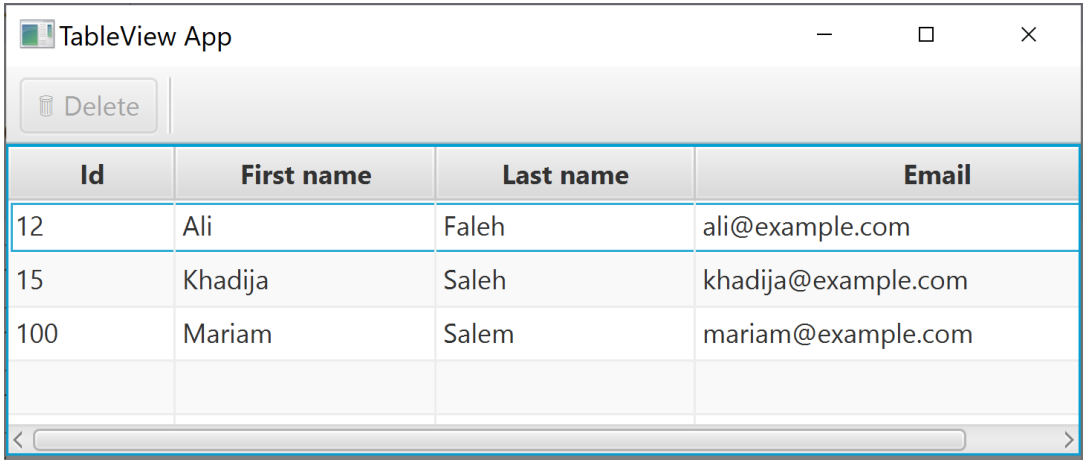
Cover Viewer App

- Binds a list of Book objects to a ListView
- When the user selects an item in the ListView, the corresponding Book's cover image is displayed in an ImageView.
  - **Property listener** is used to display the correct image when the user selects an item from the ListView



TableView

```
@FXML private TableView<Student> studentsTable;
@FXML private TableColumn<Student, Integer> idCol;
@FXML private TableColumn<Student, String> firstNameCol;
private ObservableList<Student> students = null;
public void initialize() {
    studentsTable.setItems(students);
    //Link table columns to student attributes
    idCol.setCellValueFactory(new PropertyValueFactory("id"));
    firstNameCol.setCellValueFactory(new
            PropertyValueFactory("firstName"));
}
```



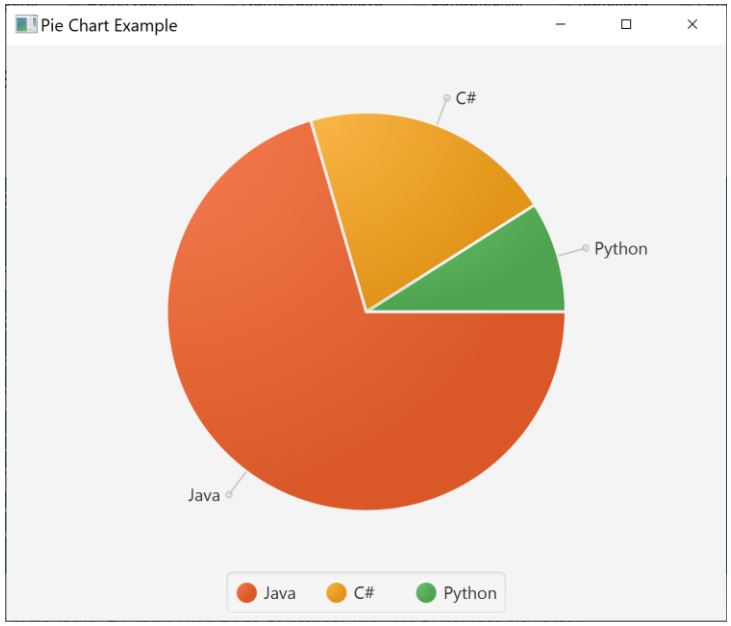
TableColumn Cell Value Factory

- A TableColumn must have a ***cell value factory*** to extracts from the object the value to be displayed in each cell (on each row) in the column.
- The PropertyValueFactory factory can extract a property value from a Java object
  - The name of the property is passed as a parameter to the PropertyValueFactory constructor:

```
PropertyValueFactory factory = new
  PropertyValueFactory<>("firstName");
```

- The property name firstName will match the getter method getFirstName() of the Person objects to get the values to display on each row.

Pie Chart




```
@FXML private PieChart pieChart;

public void initialize() {
    pieChart.setData( Model.getChartData() );
}
```

```
public class Model {
 public static ObservableList<Data> getChartData() {
    ObservableList<Data> data =
        FXCollections.observableArrayList();
    data.add(new Data("Java", 70.5));
    data.add(new Data("C#", 20.5));
    data.add(new Data("Python", 9));
    return data;
  }
}
```

Multi-scenes / Multi-windows App

Summary

- JavaFX provides a set of UI components to ease building GUI applications.
- The key expected learning outcome is gaining a good understanding and some hands on experience with:
  - UI components
  - Layout panes
  - UI event handlers
  - Building GUI Applications using the Model-view-controller (MVC) Pattern
  - Properties and Bindings

Resources

- JavaFX Tutorial

https://code.makery.ch/library/javafx-tutorial/

- Video Tutorials

https://www.youtube.com/playlist?list=PLoodc-fmtJNYbs-gYCdd5MYS4CKVbGHv2

- Scene Builder Guide

https://docs.oracle.com/javafx/scenebuilder/1/user_guide/jsbpub-user_guide.htm

https://www.youtube.com/playlist?list=PLpFneQZCNR2ktqseX11XRBc5Kyzdg2fbo

- A curated list of awesome JavaFX libraries, books, frameworks, etc...

https://github.com/mhrimaz/AwesomeJavaFX