



CMPS 251

Read Chapter 13



Graphical User Interfaces (GUI)

Dr. Abdelkarim Erradi
CSE@QU






Outline

- Common JavaFX UI Components
- Properties and Bindings
- Dialog Boxes
- Custom Dialog Box



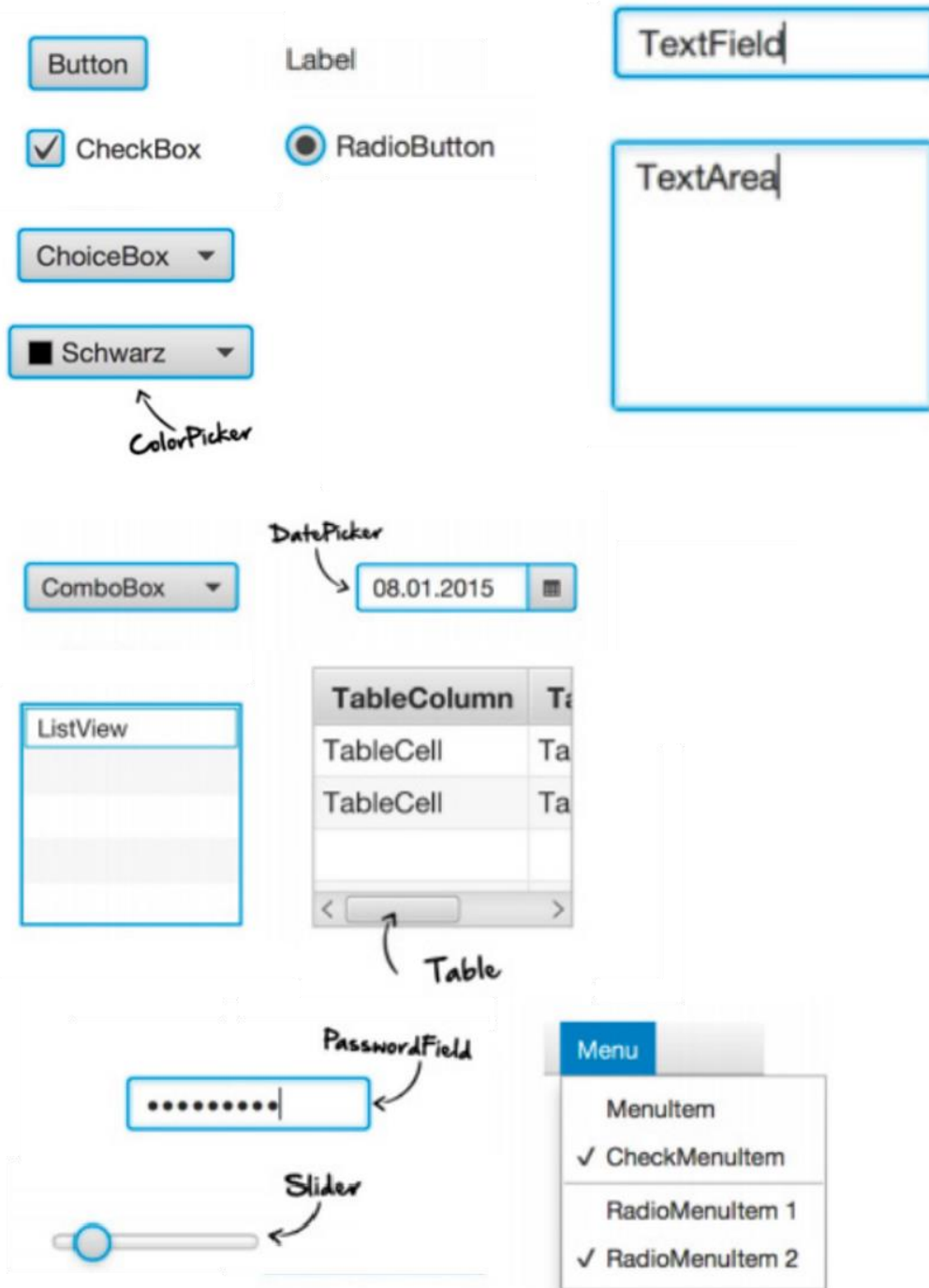
Common JavaFX UI Components

EXAMPLES

- >  > _6.controls.combobox
- >  > _6.controls.listview
- >  > _6.controls.piechart
- >  > _6.controls.qbookapp
- >  > _6.controls.registrationform

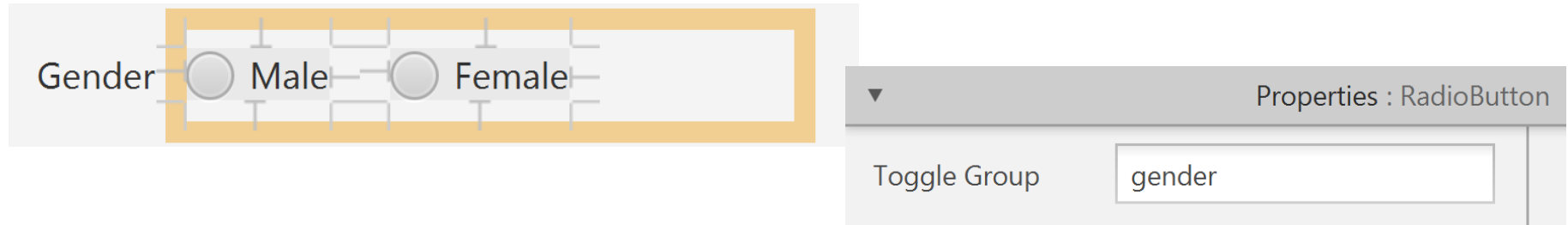
Commonly used JavaFX UI Components

- See posted examples ...



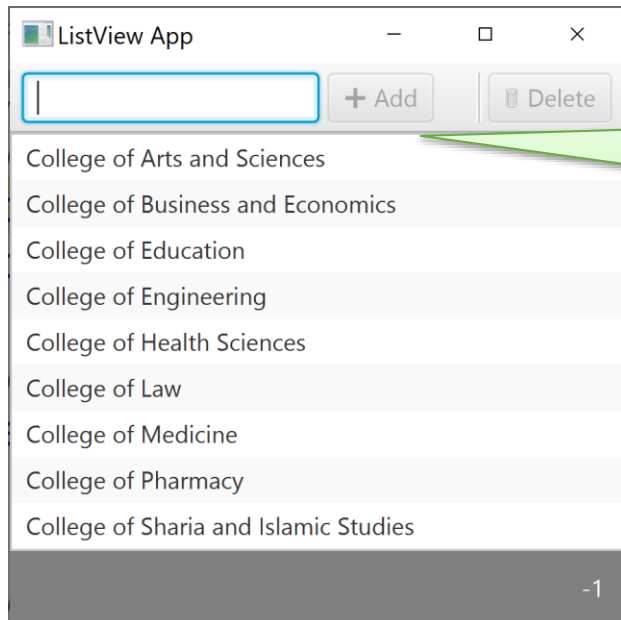
Radio Button


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



Fill a ListView using an ObservableList

```
@FXML private ListView<String> collegesListView;  
@FXML private Button deleteButton;  
  
public void initialize() {  
    ObservableList<String> collegesOL =  
        FXCollections.observableArrayList(CollegeRespository.getColleges());  
    collegesListView.setItems(collegesOL);  
}  
  
void handleAdd(ActionEvent event) {  
    collegesOL.add(college);  
}
```



The **ObservableList** notifies  the ListView of any changes (e.g., a new value is added) so that the ListView can auto-update its content



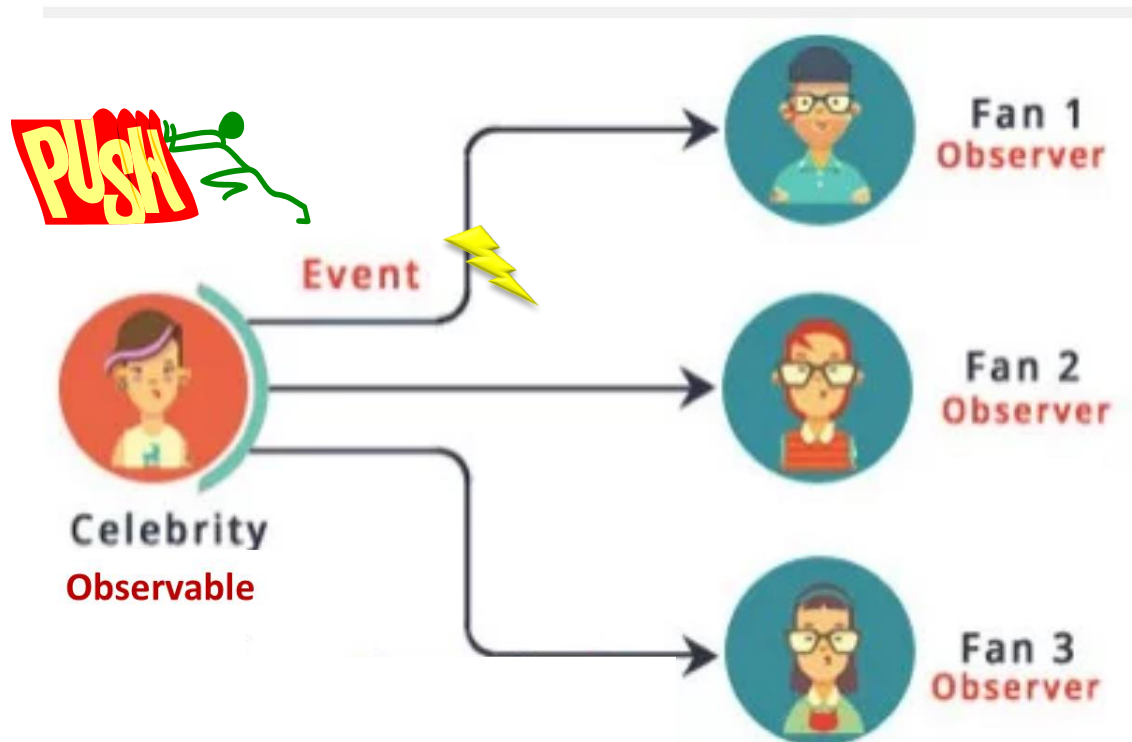
JavaFX **automatically updates the UI** whenever the observable list is updated

ObservableList

- The previous example uses a ListView control to display a list of college names
- To fill a ListView we pass an ObservableList object to the ListView using **setItems** method
 - If we make changes to the ObservableList, its **observer** (the ListView in this app) will automatically be notified of those changes
- **FXCollections.observableArrayList** static method can be used to create an ObservableList from a List

Observable - Real-Life Example

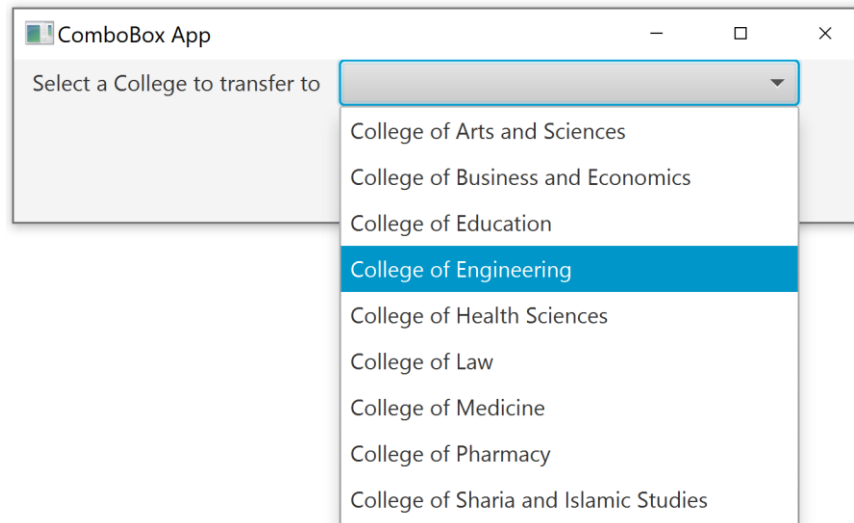
- A celebrity who has many fans on Instagram. Fans want to get all the latest updates (photos, videos, posts etc.). Here fans are **Observers** and celebrity is an **Observable**



Fill a ComboBox using an ObservableList

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

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



Getting the Selected Item

- To get a selected item from a ComboBox, ListView of a TableView you can use:

```
getSelectionModel().getSelectedItem();
```

e.g.,

```
continentCombo.getSelectionModel().getSelectedItem();
```

- To get the selected index then use:

```
continentCombo.getSelectionModel().getSelectedIndex();
```

TableView

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

Id	First name	Last name	Email
12	Ali	Faleh	ali@example.com
15	Khadija	Saleh	khadija@example.com
100	Mariam	Salem	mariam@example.com

TableColumn Cell Value Factory

- A TableColumn must have a *cell value factory* to extract the values of the object attribute to be displayed in each cell column.
- The **PropertyValueFactory** can extract a property value from a Java object
 - The name of the object attribute 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

Book 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

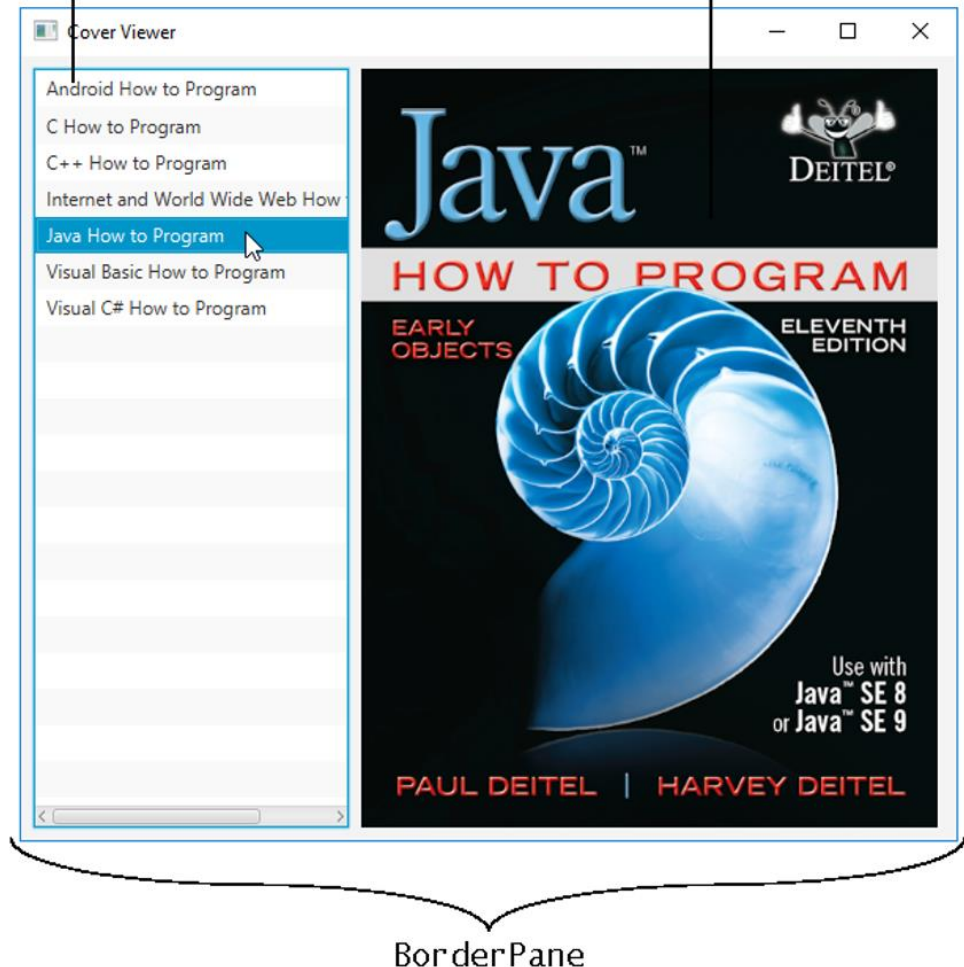
EXAMPLE



> `_6.controls.qbookapp`

booksListView

coverImageView

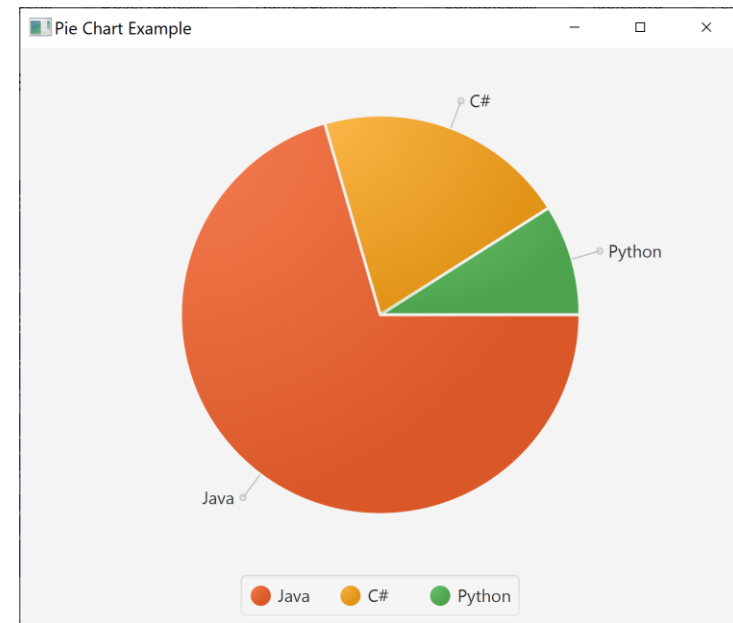


Pie Chart

```
@FXML private PieChart pieChart;

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

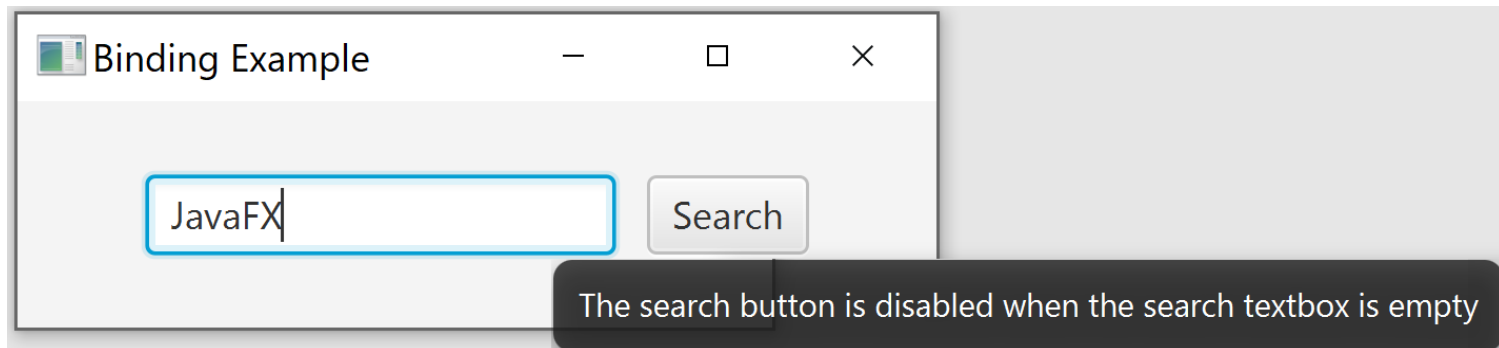
public class Model {
    public static ObservableList<Data> getChartData() {
        ObservableList<Data> data =
            FXCollections.observableArrayList();
        data.add(new Data("Java", 70.5));
        data.add(new Data("C#", 20.5));
        data.add(new Data("Python", 9));
        return data;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



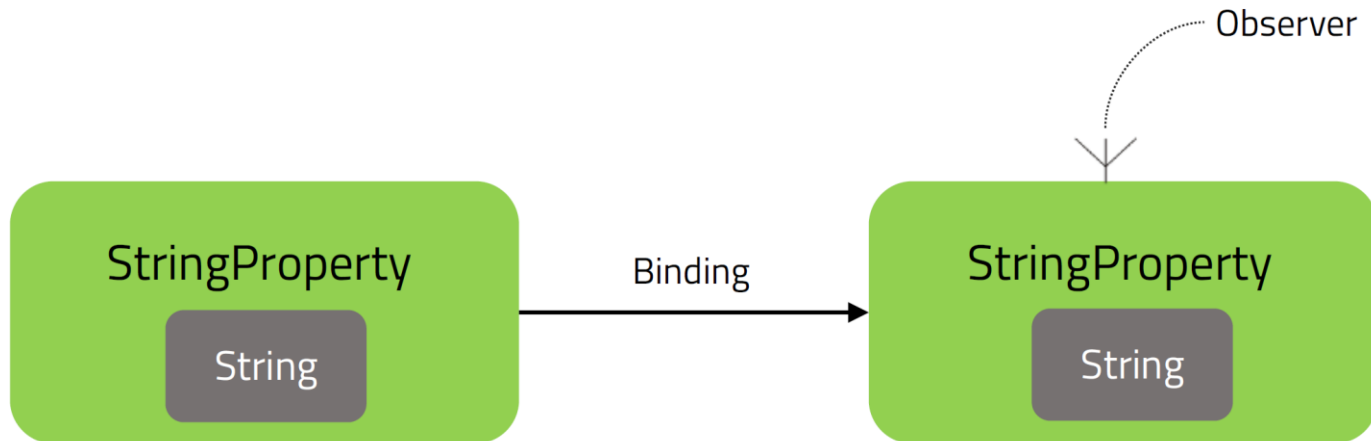
Tool Tips

- 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



Properties

- A JavaFX *property* is an object that holds a value
 - So, instead of holding an `int` value in integer primitive type, we store it in a **property** object of type `IntegerProperty`
- Most attributes of JavaFX classes are defined as properties, such as the `textProperty` of a `TextField`
 - E.g., `nameTextField.textProperty()`
 - a `ChangeListener` can be registered with an object's property to monitor its old and new values
- A property is **observable**: when a property's **value changes**, listing **objects get notified** and can respond accordingly
 - Properties fire **property change** events to registered listeners
- A key benefit of properties is **property binding**

Converting Class Attributes to Properties

- Change the data type of attributes to **property data type**:
 - String -> StringProperty
 - int -> IntegerProperty ...
- Change the constructor method to **instantiate and initialize the class properties**. E.g.,

```
this.code = new SimpleStringProperty(code);
```

- Change the getters and setters to **get/set the property value**. E.g.,

```
public String getCode() {  
    return code.getValue();  
}
```

```
public void setCode(String code) {  
    this.code.setValue(code);  
}
```

- Add public **property getter methods** consisting of the property name followed by the word “**Property**”

```
public StringProperty codeProperty() { return code; }
```

Property Binding

- Property binding enables propagating changes
 - The target listens for changes in the source and updates itself when the source changes
 - 1-Way binding syntax: **target.bind(source);** e.g.,
`label.textProperty().bind(slider.valueProperty().asString("%.0f"));`



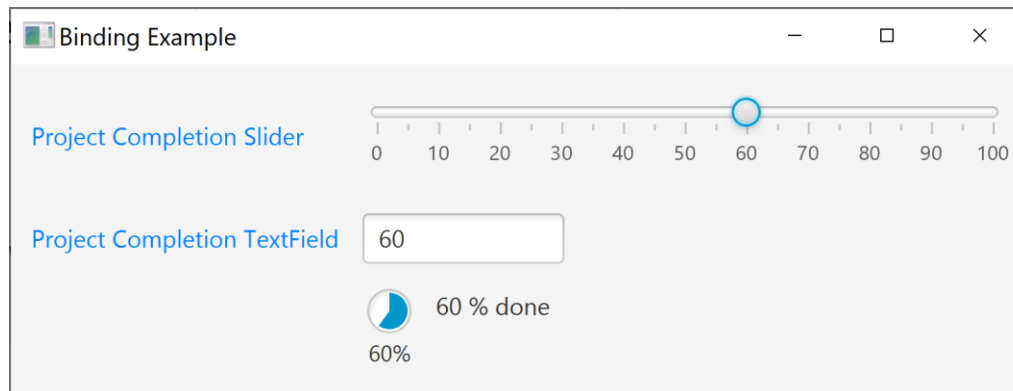
EXAMPLE

>



>

`_8.binding.aslider`



Property Binding



- Binding lets you **succinctly** **express dependencies** among object properties without registering change listeners
 - When `slider.valueProperty` is **bound** to `label.textProperty` then changes to the slider value will be reflected in the label text
 - Property Binding is used to **synchronize** the UI with associated objects
- Property binding is a better alternative than listening to the slider value change events

```
slider.valueProperty().addListener((observable, oldVal, newVal) -> {  
    label.setText(  
        String.format("%3.0f %% done", newVal.doubleValue()) );  
});
```

Unidirectional vs. Bidirectional Binding

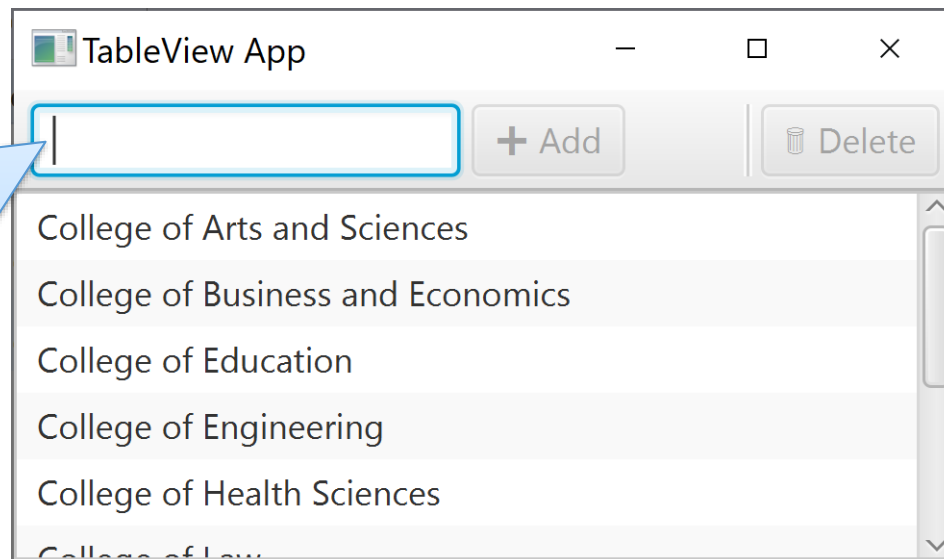
- Property-to-Property Bindings can be either Unidirectional (1-Way) vs. Bidirectional (2-Way)
- For example if a `TextField` `textProperty` and `Slider` `valueProperty` are **bound bidirectionally** then any changes of the `TextField` `text` or the `Slider` `value` will be synchronized
- On the other hand, if a `Label` `textProperty` is **bound unidirectionally** to a `Slider` `valueProperty` then only changes in the `Slider` `value` will be reflected in the `Label` `text`.

Property Binding Examples

```
//If no college selected then disable the delete button  
deleteButton.disableProperty().bind( Bindings.isNull(  
    collegesList.getSelectionModel().selectedItemProperty()) );  
}
```

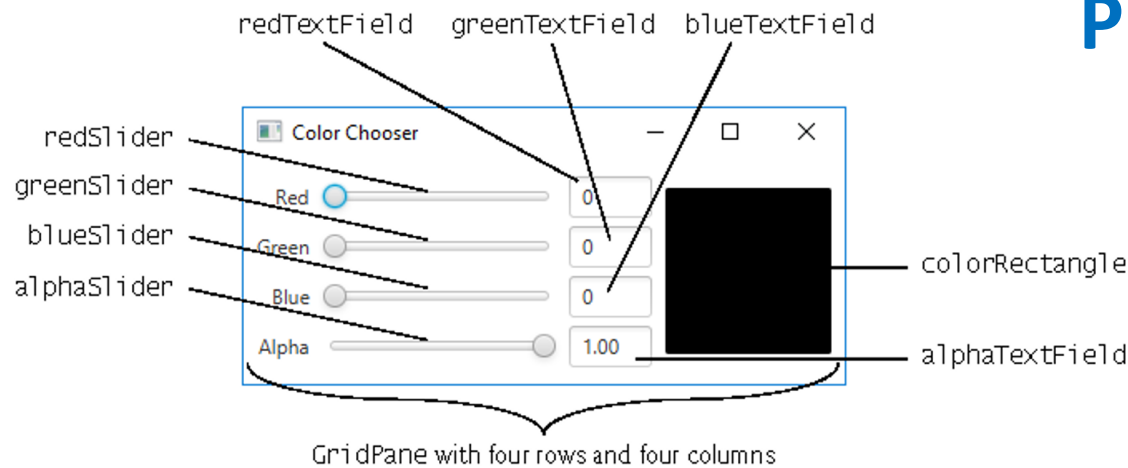
```
//If the collegeTextField is empty then disable the add button  
addButton.disableProperty().bind( Bindings.isEmpty(  
    collegeTextField.textProperty()) );
```

If the collegeTextField is empty then disable the add button




When no college is selected then the delete button should be disabled

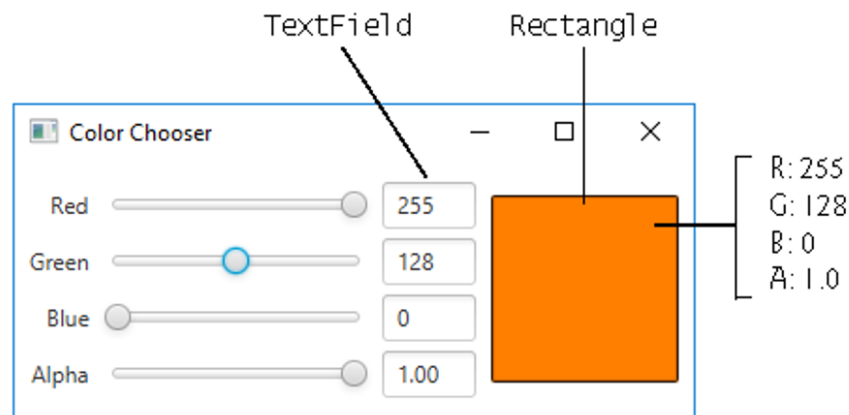
Property-to-Property Bindings Example



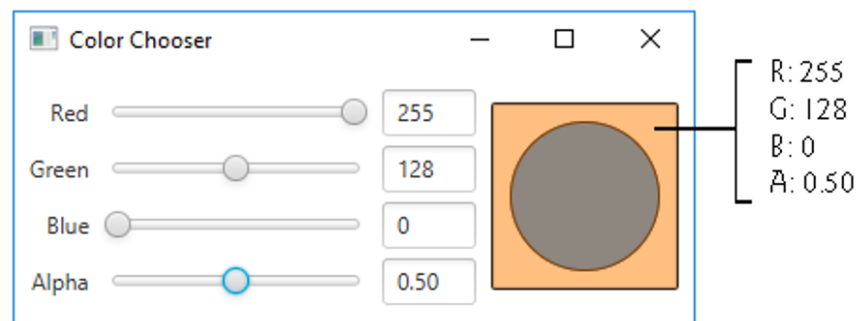
EXAMPLE

>  > `_8.binding.colorapp`

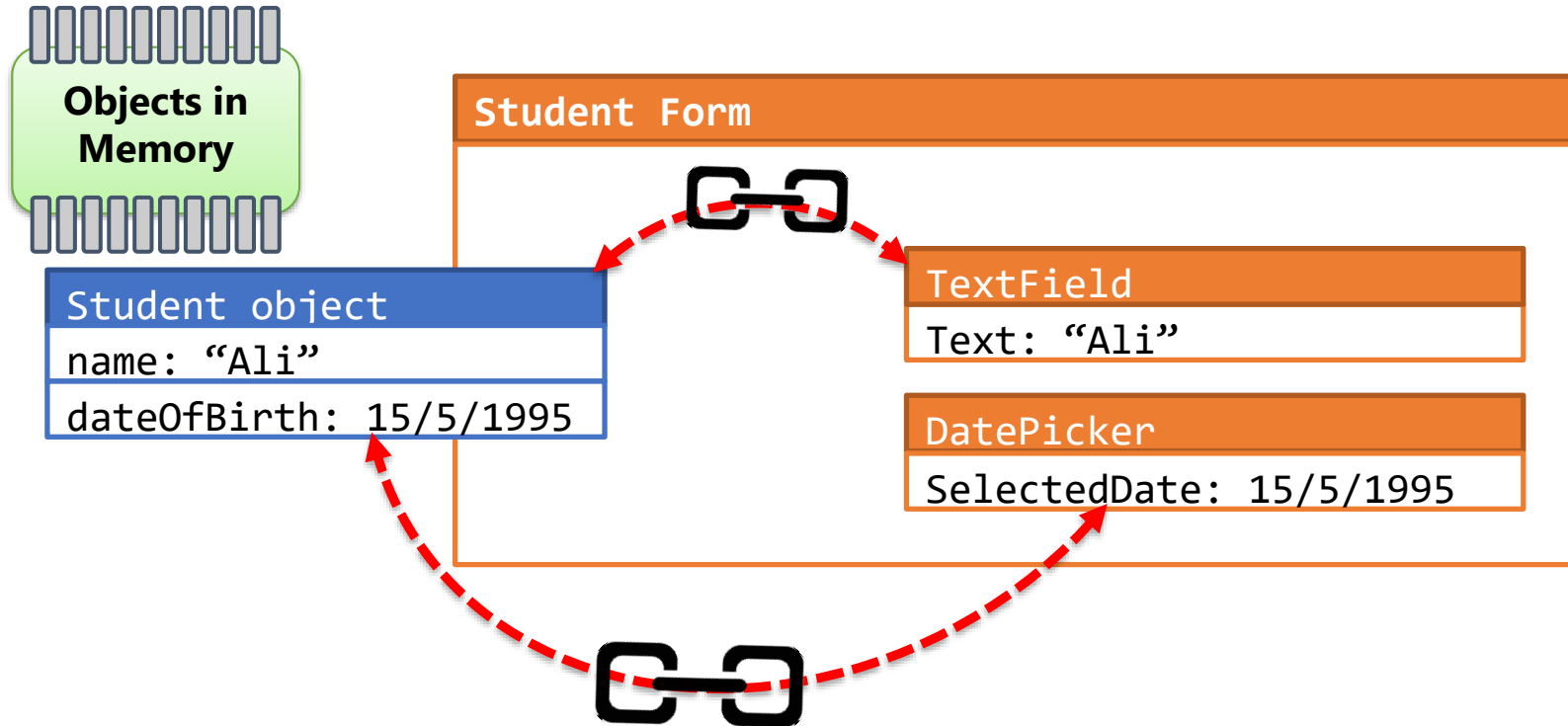
a) Using the **Red** and **Green** Sliders to create an opaque orange color



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



Two-way binding to UI components



Automatic propagation
of data changes
between the **model** and
the **view**

Binding UI components to Object Properties

EXAMPLE



> _8.binding.studentapp


```
public class Student {
    private final SimpleStringProperty firstName;
    private final SimpleStringProperty lastName;
    ...
    public SimpleStringProperty lastNameProperty()
    { return lastName; }
    public SimpleStringProperty emailProperty()
    { return email; }
    ...
}

//Bind the student properties to the UI controls
firstNameField.textProperty().bindBidirectional(
    student.firstNameProperty() );
lastNameField.textProperty().bindBidirectional(
    student.lastNameProperty());
....
```



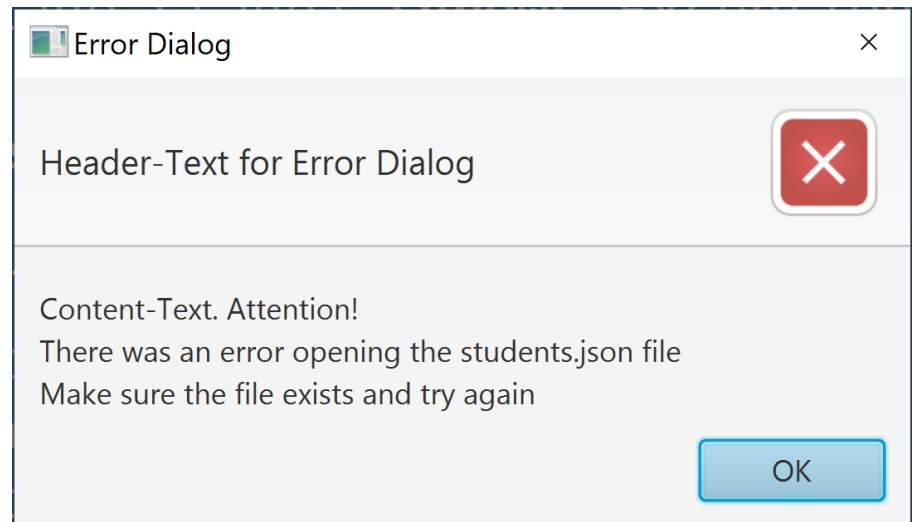
Dialog Boxes



>  > _4.dialogs

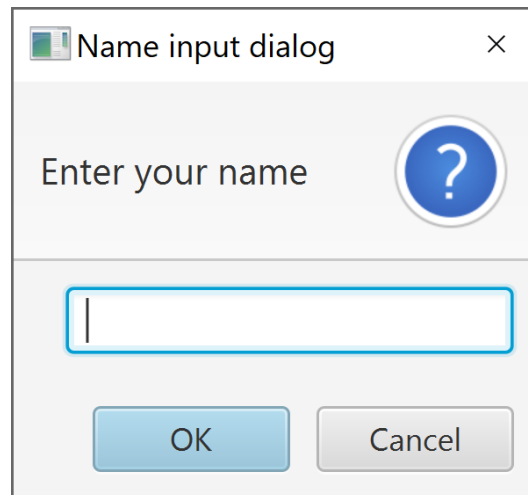
Info/Warn/Error Dialog

```
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();
}
```



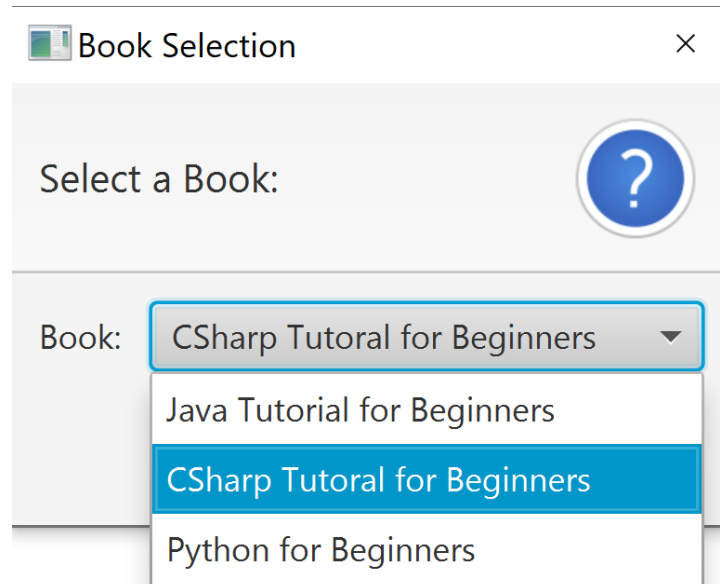
Input Dialog

```
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())) );
```



Custom Dialog Box

Create a DialogPane using SceneBuilder

EXAMPLE



> `_8.binding.studentapp`

The screenshot displays the SceneBuilder interface for creating a DialogPane. The central canvas shows a dialog box with four input fields labeled 'Id', 'First name', 'Last name', and 'Email', and 'OK' and 'Cancel' buttons at the bottom. The left 'Hierarchy' pane shows the component tree: DialogPane (containing 'insert HEADER', a GridPane (2 x 4) with four Labels and four TextFields, and 'insert CONTENT'). The right 'Properties' pane shows settings for the selected DialogPane, including 'Expanded' (checked), 'Button Types' (OK and CANCEL), and various node properties like 'Disable', 'Opacity', 'Node Orientation' (INHERIT), 'Visible' (checked), 'Focus Traversable', and 'Cache Shape' (checked).

Open the DialogPane when Add/Update button is clicked

```
//Load the fxml file and create a new popup dialog.
FXMLLoader fxmlLoader = new FXMLLoader();

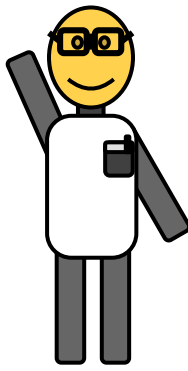
fxmlLoader.setLocation(getClass().getResource("StudentView.fxml"));
DialogPane studentDialogPane = fxmlLoader.load();

//Get the student controller associated with the view
StudentController studentController = fxmlLoader.getController();
//Pass the new student / student to the update the controller
associated with the studentDialogPane
studentController.setStudent(student);
Dialog<ButtonType> dialog = new Dialog<>();
dialog.setDialogPane(studentDialogPane);
dialog.setTitle("Add new student");

Optional<ButtonType> clickedButton = dialog.showAndWait();

if (clickedButton.get() == ButtonType.OK) {
    System.out.println("User selected ok");
} else {
    System.out.println("User selected cancel");
}
```


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>