

# JavaFX Tutorial

---

Tom Schindl <[tom.schindl@bestsolution.at](mailto:tom.schindl@bestsolution.at)>

# Intro

---

# Anatomy of an FX-App

---

# Anatomy of an FX-App

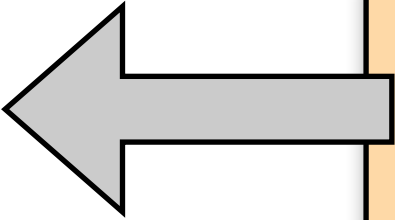
---

```
import javafx.application.Application;
```

```
public class Main extends Application {
```

```
@Override
```

```
public void start(Stage primaryStage) {
```



A large grey arrow points from the `Application` class in the code to the `Main` class, indicating inheritance. To the right of the arrow is an orange box with the text "Derived from base class".

Derived from  
base class

# Anatomy of an FX-App

---

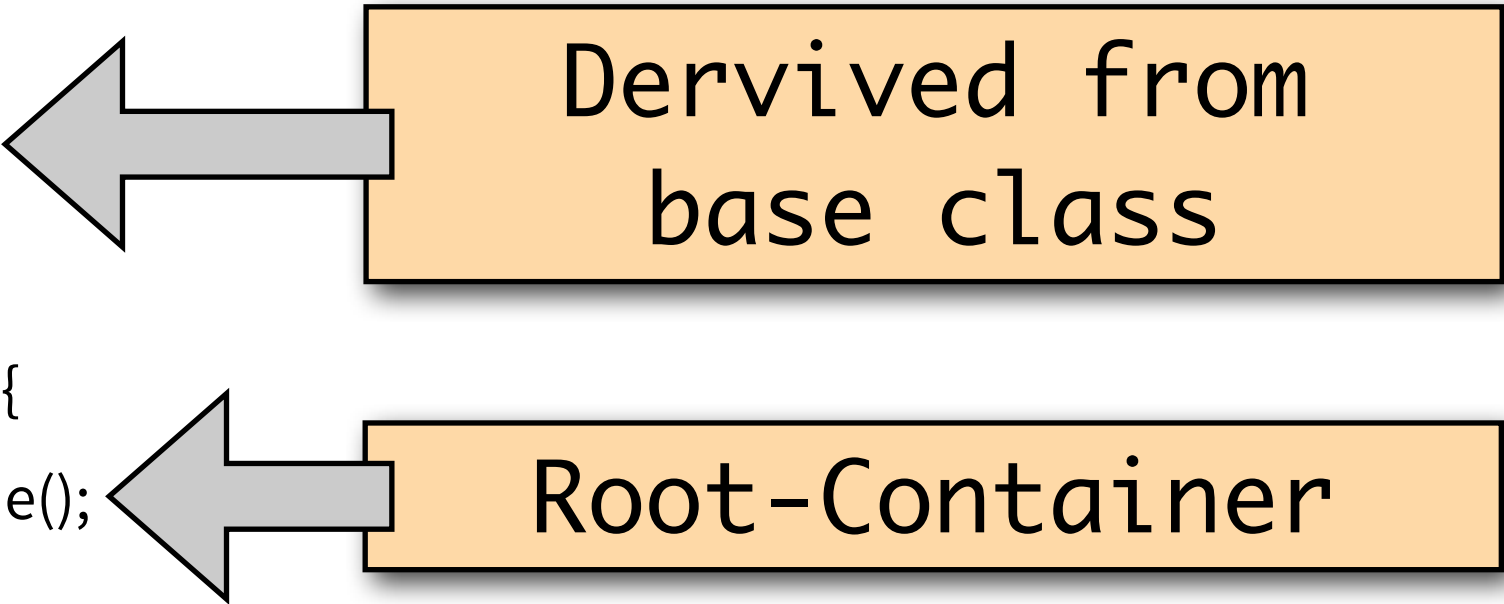
```
import javafx.application.Application;
```

```
public class Main extends Application {
```

```
@Override
```

```
public void start(Stage primaryStage) {
```

```
    BorderPane root = new BorderPane();
```



Derived from  
base class

Root-Container

# Anatomy of an FX-App

---

```
import javafx.application.Application;
```

```
public class Main extends Application {
```

```
@Override
```

```
public void start(Stage primaryStage) {
```

```
    BorderPane root = new BorderPane();
```

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



Derived from  
base class

Root-Container

Scene with size

# Anatomy of an FX-App

---

```
import javafx.application.Application;
```

```
public class Main extends Application {
```

```
@Override
```

```
public void start(Stage primaryStage) {
```

```
    BorderPane root = new BorderPane();
```

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

```
    primaryStage.setScene(scene);  
    primaryStage.show();
```

Derived from  
base class

Root-Container

Scene with size

Display

# Anatomy of an FX-App

```
import javafx.application.Application;
```

```
public class Main extends Application {
```

```
@Override
```

```
public void start(Stage primaryStage) {
```

```
    BorderPane root = new BorderPane();
```

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

```
    primaryStage.setScene(scene);  
    primaryStage.show();
```

```
public static void main(String[] args) {  
    launch(args);  
}
```

Derived from  
base class

Root-Container

Scene with size

Display

inherited method



# Lab HelloWorld

- Setting up Eclipse
- Creating your first JavaFX project
- Attaching the first Event-Listener

# Lab Hello World

---

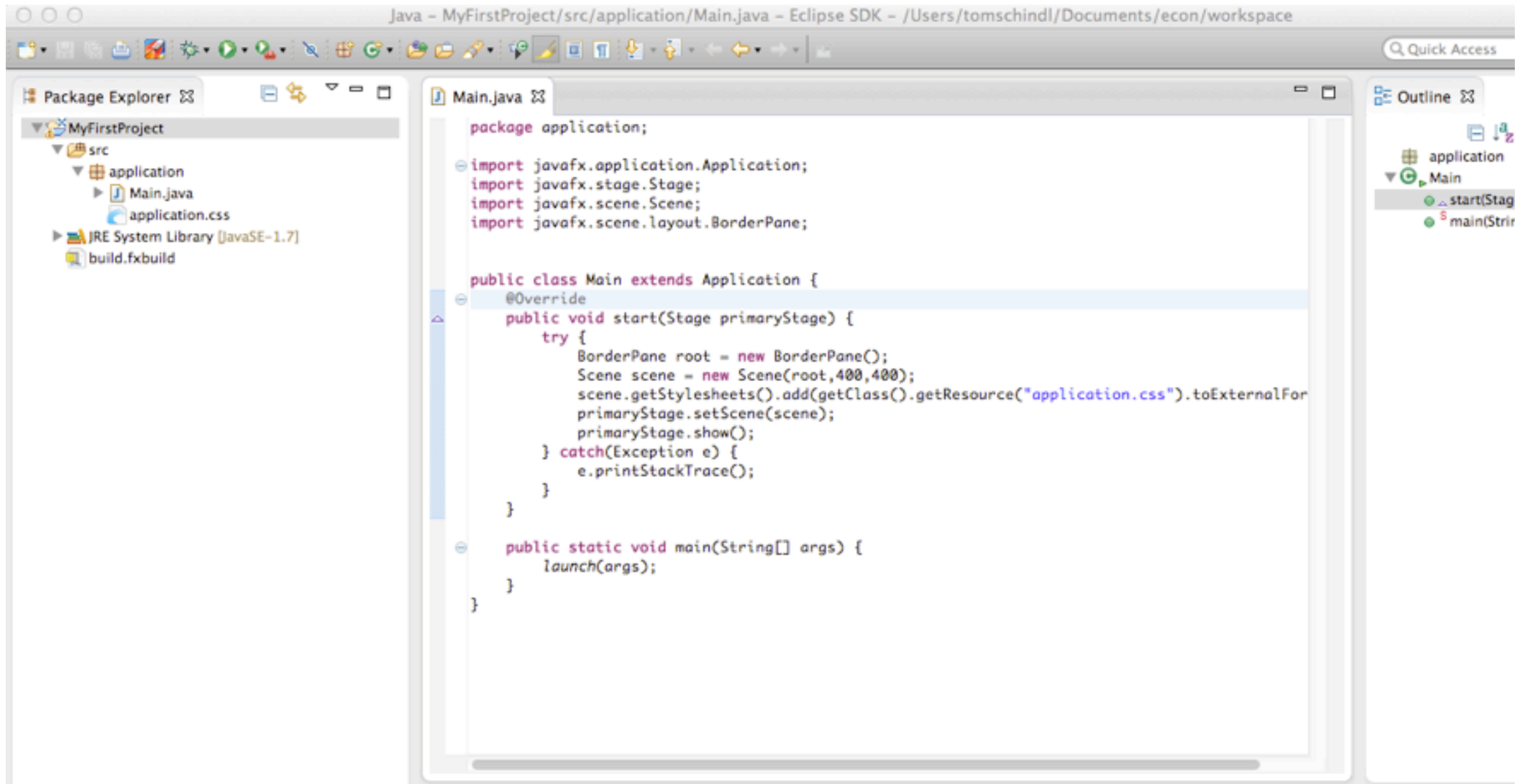
- ▶ Create a directory named „fx\_tutorial“ on your filesystem e.g. `C:\fx_tutorial`, `/Users/tom/fx_tutorial`
- ▶ Move `eclipse-SDK-4.3.0-$arch$.tar.gz/.zip` to the directory and uncompress it there
- ▶ Install JDK8u111
  - ▶ Linux: extract it next to your eclipse-SDK
- ▶ Launch Eclipse with JDK8
  - ▶ Linux: Launch with `./eclipse -vm ../jdk8.../`
  - ▶ Check that JDK8 is used via About > Installation Details > Configuration - search for „eclipse.vm“

# Lab Hello World

---

- ▶ File > New > Project ...
- ▶ Search for the JavaFX category
- ▶ Select „JavaFX Project“ > Next
- ▶ Enter the following data:
  - ▶ Project name: MyFirstProject
  - ▶ Use an execution environment JRE: JavaSE-1.7
- ▶ Select: Finish

# Lab Hello World



# Lab Hello World

---

- ▶ Create an instance of `javafx.scene.control.Button` which displays a text „Hello World!“
- ▶ Handle a button click and print „Hello World!“
  - ▶ Try to use the `setOnAction` API
  - ▶ Try to use the `addEventHandler` API
- ▶ Display the button in the center of the `BorderPane`

# Lab Hello World

---

```
BorderPane root = new BorderPane();
Button b = new Button("Hello World");
b.setOnAction(new EventHandler<ActionEvent>() {

    @Override
    public void handle(ActionEvent event) {
        System.out.println("Hello World via setOnAction!");
    }
});
b.addEventHandler(ActionEvent.ACTION, new EventHandler<ActionEvent>() {

    @Override
    public void handle(ActionEvent event) {
        System.out.println("Hello World via addEventHandler!");
    }
});
root.setCenter(b);
```

# FX-Properties

# FX-Properties

---

- ▶ JavaFX Beans use extend the JavaBean pattern
  - ▶ `getName$/setName$` method
  - ▶ `$name$Property` method
- ▶ `property-method` returns
  - ▶ `read/writable`: `javafx.beans.property.Property`
  - ▶ `readonly`: `javafx.beans.property.ReadOnlyProperty`
- ▶ `Property-Objects` are observable and can be bound together



# FX-Properties

---

```
public class JavaBean {  
    private String name;  
  
    private PropertyChangeSupport support = new PropertyChangeSupport(this);  
  
    public void setName(String name) {  
        support.firePropertyChange("name", this.name, this.name = name);  
    }  
  
    public String getName() {  
        return this.name;  
    }  
}
```

# FX-Properties

---

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

# FX-Properties

---

- ▶ Properties can be bound
  - ▶ Unidirectional: `Property#bind()`
  - ▶ Bidirectional: `Property#bindBidirectional()`
- ▶ Unlink bindings:
  - ▶ Unidirectional: `Property#unbind()`
  - ▶ Bidirectional: `Property#unbindBidirectional()`

# Lab FXProperties

- Create JavaFX Bean
- Create UI with and bind properties

# Lab FXProperties

---

- ▶ Create a new JavaFX-Project
- ▶ Create a JavaFX Bean
  - ▶ Name: MyBean
  - ▶ Properties: String-Property named „text“
- ▶ Add the following UI-Elements to the Main class
  - ▶ top: javafx.scene.control.TextField
  - ▶ center: javafx.scene.text.Text
  - ▶ left: javafx.scene.control.Slider (hint: orientation!)
  - ▶ right: javafx.scene.control.Slider

# Lab FXProperties

---

- ▶ Make the slider accept values in range min=1 & max=10
- ▶ Create an instance of MyBean
- ▶ Bind:
  - ▶ `bidirectional: MyBean#text to TextField#text`
  - ▶ `unidirectional:`
    - ▶ `MyBean#text to Text#text`
    - ▶ `H-Slider#value to Text#scaleX`
    - ▶ `V-Slider#value to Text#scaleY`

# Lab FXProperties (for the fast one)

---

- ▶ Make sure the sliders are only modifiable when the text field has a value entered

# FX-Layouts



# FX Layouts

---

- ▶ JavaFX comes with predefined layout panes like
  - ▶ `javafx.scene.layout.BorderPane`
  - ▶ `javafx.scene.layout.HBox`
  - ▶ `javafx.scene.layout.VBox`
  - ▶ `javafx.scene.layout.GridPane`
- ▶ Layout constraints are applied through constant setters

```
BorderPane root = new BorderPane();  
Button child = new Button("Layout Test");  
BorderPane.setAlignment(child, Pos.CENTER_LEFT);  
root.setCenter(child);
```

# FX Layouts

---

- ▶ Additional layouts
  - ▶ SWT-Layouts part of e(fx)clipse
    - ▶ `org.eclipse.fx.ui.panes.GridLayoutPane`
    - ▶ `org.eclipse.fx.ui.panes.FillLayoutPane`
    - ▶ `org.eclipse.fx.ui.panes.RowLayoutPane`
  - ▶ MigPane (<http://www.miglayout.com/>)

# FXML

# FXML

---

- ▶ FXML is a declarative way to define a JavaFX-Scenegraph
- ▶ WYSIWYG Tool called SceneBuilder
- ▶ Rules how to map Java to XML-Constructors
  - ▶ classes get xml-elements  
Java: `Button b = new Button();`  
FXML: `<Button>`
  - ▶ simple attribute types get xml-attributes  
Java: `b.setText("Hello World");`  
FXML: `<Button text="Hello World"`
  - ▶ complex attribute types get xml-elements  
Java: `new BorderPane().setCenter(new Button("Hello World"))`  
FXML: `<BorderPane><center><Button text="Hello World" /></center></BorderPane>`

# FXML

---

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<?import javafx.scene.layout.HBox?>
```

```
<?import javafx.scene.control.Button?>
```

```
import javafx.scene.control.Button;
```

```
import javafx.scene.layout.HBox;
```

```
<HBox xmlns:fx="http://javafx.com/fxml">
```

```
    <children>
```

```
        <Button
```

```
            text="Hello World">
```

```
        </Button>
```

```
    </children>
```

```
</HBox>
```

```
HBox box = new HBox();
```

```
Button button = new Button("Hello World");
```

```
box.getChildren().add(button);
```

# FXML

---

## ► Executing actions

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<?import javafx.scene.layout.BorderPane?>
```

```
<?import javafx.scene.layout.HBox?>
```

```
<?import javafx.scene.control.Button?>
```

```
<HBox xmlns:fx="http://javafx.com/fxml"
```

```
  fx:controller="application.SampleController">
```

```
  <children>
```

```
    <Button
```

```
      fx:id="mybutton"
```

```
      text="Hello World"
```

```
      onAction="#run">
```

```
    </Button>
```

```
  </children>
```

```
</HBox>
```

# FXML

---

## ► Executing actions

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<?import javafx.scene.layout.BorderPane?>
```

```
<?import javafx.scene.layout.HBox?>
```

```
<?import javafx.scene.control.Button?>
```

```
<HBox xmlns:fx="http://javafx.com/fxml"
```

```
  fx:controller="application.SampleController">
```

```
  <children>
```

```
    <Button
```

```
      fx:id="mybutton"
```

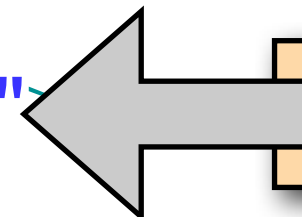
```
      text="Hello World"
```

```
      onAction="#run">
```

```
    </Button>
```

```
  </children>
```

```
</HBox>
```



Java-Class

# FXML

---

## ► Executing actions

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<?import javafx.scene.layout.BorderPane?>
```

```
<?import javafx.scene.layout.HBox?>
```

```
<?import javafx.scene.control.Button?>
```

```
<HBox xmlns:fx="http://javafx.com/fxml"
```

```
  fx:controller="application.SampleController">
```

```
  <children>
```

```
    <Button
```

```
      fx:id="mybutton"
```

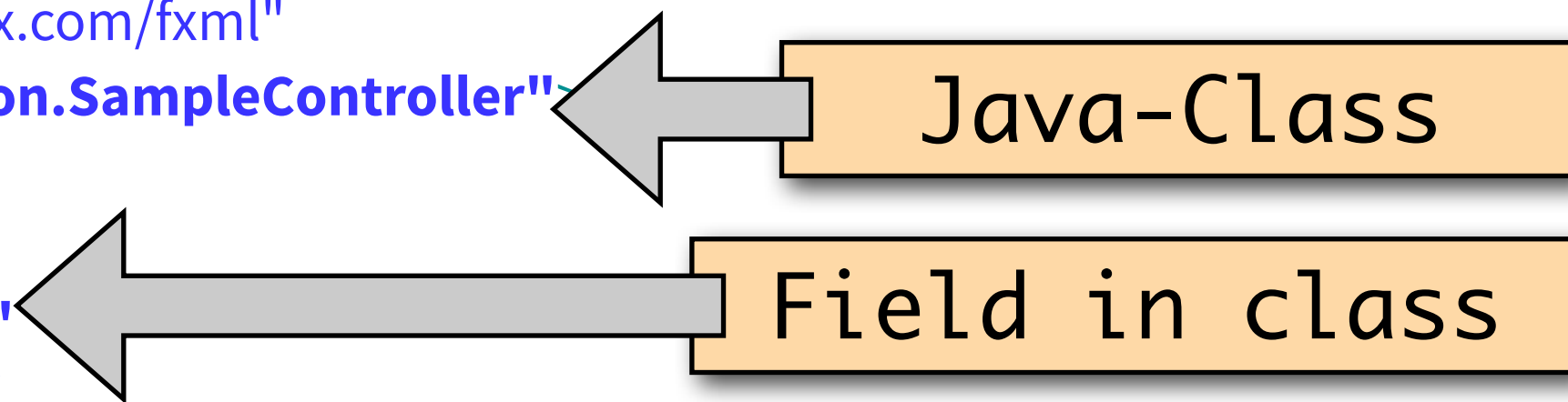
```
      text="Hello World"
```

```
      onAction="#run">
```

```
    </Button>
```

```
  </children>
```

```
</HBox>
```



Java-Class

Field in class



# FXML

## ► Executing actions

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<?import javafx.scene.layout.BorderPane?>
```

```
<?import javafx.scene.layout.HBox?>
```

```
<?import javafx.scene.control.Button?>
```

```
<HBox xmlns:fx="http://javafx.com/fxml"
```

```
  fx:controller="application.SampleController">
```

```
  <children>
```

```
    <Button
```

```
      fx:id="mybutton"
```

```
      text="Hello World"
```

```
      onAction="#run">
```

```
    </Button>
```

```
  </children>
```

```
</HBox>
```

Java-Class

Field in class

Method in class

# FXML

---

## ► Executing actions / accessing stuff in Java

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<?import javafx.scene.layout.BorderPane?>
```

```
<?import javafx.scene.layout.HBox?>
```

```
<?import javafx.scene.control.Button?>
```

```
<HBox xmlns:fx="http://javafx.com/fxml"  
      fx:controller="application.SampleController">
```

```
  <children>
```

```
    <Button
```

```
      fx:id="mybutton"
```

```
      text="Hello World"
```

```
      onAction="#run">
```

```
    </Button>
```

```
  </children>
```

```
</HBox>
```

```
package application;
```

```
import javafx.fxml.FXML;
```

```
import javafx.scene.control.Button;
```

```
public class SampleController {  
    @FXML Button mybutton;
```

```
    @FXML
```

```
    public void run() {
```

```
    }
```

```
}
```

# FXML

---

- ▶ layout-constraint support

- ▶ simple constraints: `<Button BorderPane.alignment="CENTER_LEFT">`
- ▶ complex constraints: `<BorderPane.margin><Insets left="10"></Insets></BorderPane.margin>`

- ▶ i18n support

- ▶ prefix value with %: `<Button fx:id="mybutton" text="%hello.world">`
- ▶ preview: `<?scenebuilder-preview-i18n-resource messages.properties?>`

- ▶ media resource support

- ▶ prefix value with @: `<Image url="@Money-icon_48.png" />`

- ▶ loading FXML-Files using `javafx.fxml.FXMLLoader.load`

# Lab FXML

- ▶ Create FXML
- ▶ Connect to controller
- ▶ Use i18n

# Lab FXML

---

- ▶ Create a JavaFX-Project named „FXMLProject“
  - ▶ Navigate to the last page in the wizard
    - ▶ Language: FXML
    - ▶ Root-Type: `javafx.scene.layout.BorderPane`
    - ▶ Filename: Sample
    - ▶ Controller Name: SampleController
- ▶ Open Preview using Window > Show View > JavaFX > JavaFX Preview

# Lab FXML

---

- ▶ Create basic UI
  - ▶ Create a center-element below the BorderPane
  - ▶ Add a button-element with a text „Hello World“
  - ▶ Align the button to CENTER\_LEFT
- ▶ Open the SampleController
- ▶ Go back to the Sample.fxml
- ▶ Add an onAction-Attribute and set #run as the value
  - ▶ Notice the error marker
  - ▶ Use auto-correction CTRL/CMD+1
  - ▶ Select first proposal and notice SampleController change

# Lab FXML

---

- ▶ Add an `fx:id` to Button-element and use value `mybutton`
  - ▶ Notice warning marker
  - ▶ Use auto-correction CTRL/CMD+1
  - ▶ Select first proposal and notice `SampleController` change
- ▶ Modify `SampleController#run` to update the `text-Value` of the button
- ▶ Create a `messages.properties-File`
  - ▶ Add a key „`hello.world`“
  - ▶ Update the FXML to use `hello.world`
  - ▶ Update the Main-Code to use `FXMLLoader.load(URL,ResourceBundle)`

# Lab FXML (for the fast ones)

---

- ▶ Try to add an image to the button
  - ▶ Hints: `graphic`, `ImageView`, `Image`
  - ▶ Hints 2: FXML-Editor does not know about `url-Property` of `Image`



# FXGraph

# FXGraph

---

- ▶ FXGraph is a declarative language with a similar notation to JSON
  - ▶ Remove a lot of noise created by XML
- ▶ It „compiles“ to FXML (=no extra runtime libs needed)
- ▶ Has some extra features
- ▶ Definitions:
  - ▶ Object-Def: `Button{}`
  - ▶ Simple-Attribute: `Button { text : "Hello World" }`
  - ▶ Complex-Attribute: `BorderPane { center : Button { text : "Hello World" } }`

# FXGraph

---

```
package application
```

```
import javafx.scene.layout.BorderPane
```

```
import application.SampleController
```

```
import javafx.scene.control.Button
```

```
component Sample resourcefile "messages.properties" controlledby SampleController {  
    BorderPane {  
        center : Button {  
            text : "Hello World"  
        }  
    }  
}
```

# FXGraph

---

```
package application
```

```
import javafx.scene.layout.BorderPane
```

```
import application.SampleController
```

```
import javafx.scene.control.Button
```

```
component Sample resourcefile "messages.properties" controlledby SampleController {
```

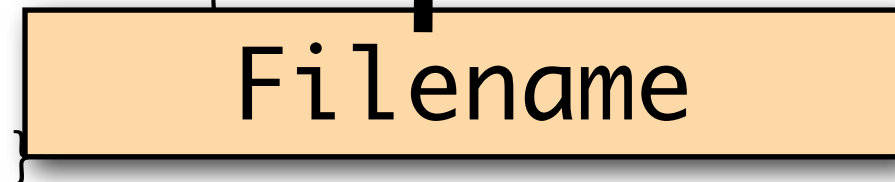
```
    BorderPane {
```

```
        center : Button {
```

```
            text "Hello World"
```

```
        }
```

```
    }
```



# FXGraph

---

```
package application
```

```
import javafx.scene.layout.BorderPane
```

```
import application.SampleController
```

```
import javafx.scene.control.Button
```

```
component Sample resourcefile "messages.properties" controlledby SampleController {
```

```
    BorderPane {
```

```
        center : Button {
```

```
            text "Hello World"
```

```
        }
```

```
    }
```

Translations

Filename

# FXGraph

```
package application
```

```
import javafx.scene.layout.BorderPane
```

```
import application.SampleController
```

```
import javafx.scene.control.Button
```

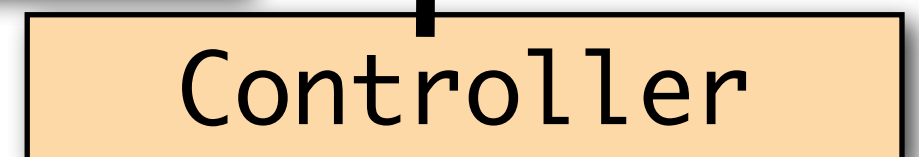
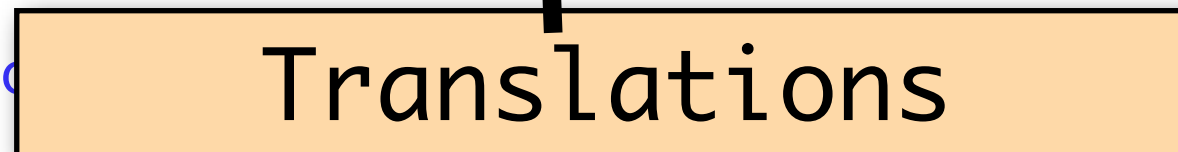
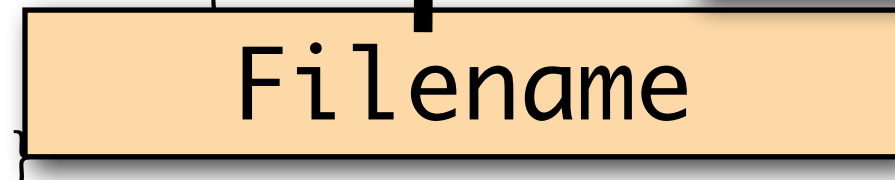
```
component Sample resourcefile "messages.properties" controlledby SampleController {
```

```
BorderPane {
```

```
center : Button {
```

```
text "Hello World"
```

```
}
```



# FXGraph

---

- ▶ Layout-constraint support:
  - ▶ simple constraints: `Button { static alignment : "CENTER_LEFT" }`
  - ▶ complex constraints: `Button { static margin : Insets { left : 10 } }`
- ▶ i18n support
  - ▶ prefix string with `rstring`: `Button { text : rstring "hello.world" }`
- ▶ media support:
  - ▶ prefix string with `location`: `Image { url : location "Money-icon_48.png" }`
- ▶ preview marker:
  - ▶ prefix an attribute with `preview`: `TextField { preview text : "Preview only" }`

# FXGraph

---

- ▶ Executing actions / accessing stuff in Java

```
component Sample controlledby application.CurrencyController {  
    BorderPane {  
        center : Button id mybutton {  
            text : "Hello World",  
            onAction : controllermethod run  
        }  
    }  
}
```



# FXGraph

---

- ▶ Executing actions / accessing stuff in Java

```
component Sample controlledby application.CurrencyController {  
    BorderPane {  
        center : Button id mybutton {  
            text : "Hello World",  
            onAction : controllermethod run  
        }  
    }  
}
```



Field in class

# FXGraph

---

- ▶ Executing actions / accessing stuff in Java

```
component Sample controlledby application.CurrencyController {  
  BorderPane {  
    center : Button id mybutton {  
      text : "Hello World",  
      onAction : controllermethod run  
    }  
  }  
}
```

Field in class



Method in class



---

# Lab FXGraph

- ▶ Create complex UI
- ▶ Connect to controller
- ▶ Use i18n

# Lab FXGraph

---

- ▶ Create a JavaFX-Project named „FXGraphProject“
  - ▶ Navigate to the last page in the wizard
    - ▶ Language: FXGraph
    - ▶ Root-Type: `javafx.scene.layout.BorderPane`
    - ▶ Filename: `Currency`
    - ▶ Controller Name: `CurrencyController`

# Lab FXGraph

---

## ► Create the UI

FXGraph

Name

Abbreviation

Add Remove

# Lab FXGraph

---

- ▶ Put another `javafx.scene.layout.BorderPane` in the `left-Property`
  - ▶ put a `javafx.scene.control.ListView` in the center
  - ▶ put a `javafx.scene.layout.HBox` in the bottom
    - ▶ add 2 `javafx.scene.control.Button` as the children
- ▶ Put `javafx.scene.layout.GridPane` in the `center Property`  
(Hint `row`, `column-index` and `hgrow` can be set using `static`)
  - ▶ add a `javafx.scene.control.Label` (`text=Name`)
  - ▶ add a `javafx.scene.control.TextField`
  - ▶ add a `javafx.scene.control.Label` (`text=Abbreviation`)
  - ▶ add a `javafx.scene.control.TextField`

# Lab FXGraph

---

- ▶ Create a file `messages.properties`
  - ▶ Add the following keys with translations:  
`common.add`  
`common.remove`  
`currency.name`  
`currency.abbrev`
  - ▶ Modify `Currency.fxgraph` adding `resourcefile "messages.properties"` in the component definition
  - ▶ Use `rstring` in the `Button` and `Label` text-property
- ▶ Connect the following to the controller (using `id`)
  - ▶ `ListView` as `currencyList`
  - ▶ `TextField` as `nameField`, `abbreviationField`

# Lab FXGraph

---

- ▶ Connect the buttons `onAction-Slot` to the controller (using `controllermethod`)
  - ▶ Add Button to `addCurrency`
  - ▶ Remove Button to `removeCurrency`
- ▶ Set the `id-attribute(!!!)` of the `GridPane` to „`currencyDetail`“



# CSS

# CSS

---

- ▶ JavaFX uses CSS to theme ALL elements
- ▶ Selectors supported are mainly CSS2 compatible
  - ▶ Element-Selectors: Applies to the classname in the SceneGraph (e.g. `BorderPane`, `HBox`, ...)
  - ▶ ID-Selectors: Applies to the id-attribute set via `Node#id: String`
  - ▶ Class-Selectors: Applies to the classes assigned through `Node#styleClass: ObservableList<String>`

# CSS

---

- ▶ JavaFX-Controls automatically assign the controls name to the Skin-Class making up the control. e.g. Button styles itself not with Button but .button

# CSS

---

- ▶ JavaFX-Controls automatically assign the controls name to the Skin-Class making up the control. e.g. Button styles itself not with Button but .button

SceneGraph

BorderPane

TitledPane

# CSS

---

- ▶ JavaFX-Controls automatically assign the controls name to the Skin-Class making up the control. e.g. Button styles itself not with Button but .button

SceneGraph

BorderPane

TitledPane

StackPane

HBox

Label

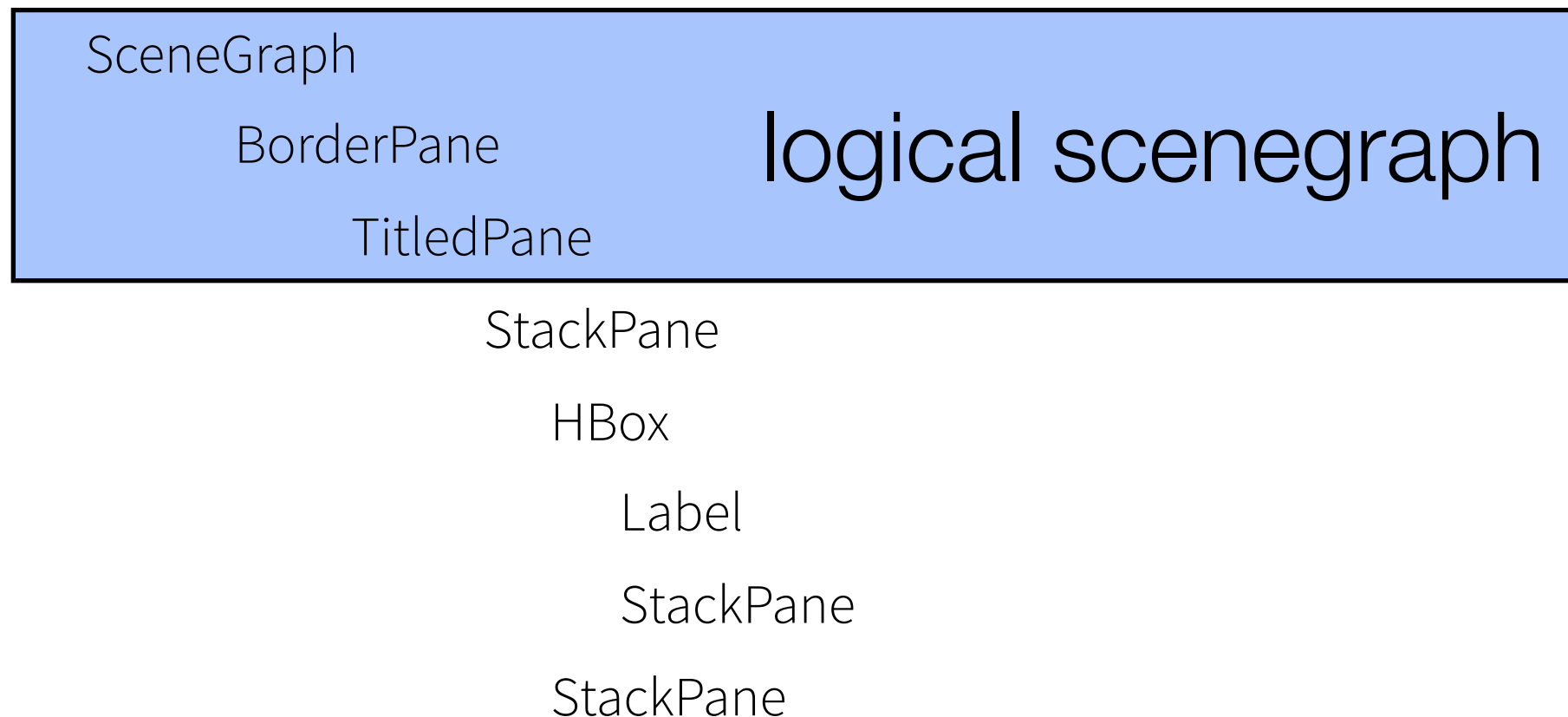
StackPane

StackPane

# CSS

---

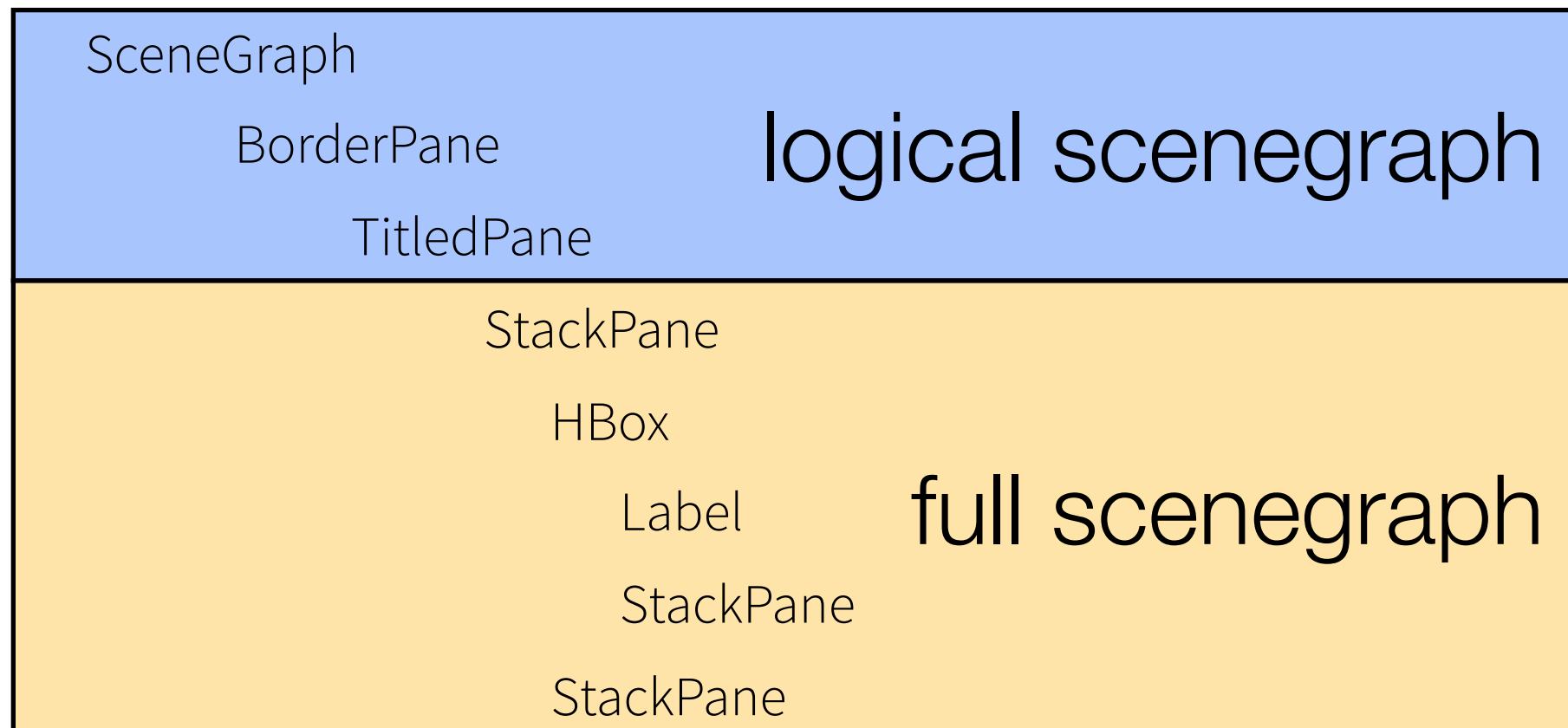
- ▶ JavaFX-Controls automatically assign the controls name to the Skin-Class making up the control. e.g. Button styles itself not with Button but .button



# CSS

---

- JavaFX-Controls automatically assign the controls name to the Skin-Class making up the control. e.g. Button styles itself not with Button but .button



# CSS

---

- ▶ JavaFX properties all start with -fx
- ▶ Informations which properties apply to which element are available from <http://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html>
- ▶ e(fx)clipse CSS-Editor knows which properties apply if you use the predefined class and element selectors



# Lab CSS

- Use some simple css

# Lab CSS

---

- ▶ Open the `application.css` in the `FXGraphProject`
  - ▶ Redefine the `hgap` / `vgap` for `GripPanels`
  - ▶ Redefine the `padding` for the `GridPanel` with ID `currencyDetail`

# Working with Views

# Working with Views

---

# Working with Views

---

- ▶ All views are virtual (cells are reused!!)

# Working with Views

---

- ▶ All views are virtual (cells are reused!!)
- ▶ All views are made up of Cell-Nodes

# Working with Views

---

- ▶ All views are virtual (cells are reused!!)
- ▶ All views are made up of Cell-Nodes
- ▶ Cell-Nodes are created through factories

# Working with Views

---

- ▶ All views are virtual (cells are reused!!)
- ▶ All views are made up of Cell-Nodes
- ▶ Cell-Nodes are created through factories

```
ListView<Currency> currencyList = new ListView<>();  
currencyList.setCellFactory(new Callback<ListView<Currency>, ListCell<Currency>>() {
```

```
    @Override  
    public ListCell<Currency> call(ListView<Currency> param) {  
        return new CurrencyCell();  
    }  
});
```



# Working with Views

---

- ▶ All views are virtual (cells are reused!!)
- ▶ All views are made up of Cell-Nodes
- ▶ Cell-Nodes are created through factories

```
ListView<Currency> currencyList = new ListView<>();  
currencyList.setCellFactory(new Callback<ListView<Currency>, ListCell<Currency>>() {
```

```
    @Override  
    public ListCell<Currency> call(ListView<Currency> param) {  
        return new CurrencyCell();  
    }  
});
```

# JDK7-Style

# Working with Views

---

- ▶ All views are virtual (cells are reused!!)
- ▶ All views are made up of Cell-Nodes
- ▶ Cell-Nodes are created through factories

```
ListView<Currency> currencyList = new ListView<>();  
currencyList.setCellFactory(new Callback<ListView<Currency>, ListCell<Currency>>() {
```

```
    @Override  
    public ListCell<Currency> call(ListView<Currency> param) {  
        return new CurrencyCell();  
    }  
});
```

```
ListView<Currency> currencyList = new ListView<>();  
currencyList.setCellFactory((param) -> new CurrencyCell());
```

# JDK7-Style

# Working with Views

---

# Working with Views

---

```
public class CurrencyCell extends ListCell<Currency> {  
    @Override  
    protected void updateItem(Currency item, boolean empty) {  
        if( item != null && ! empty ) {  
            setText(item.getName());  
        } else {  
            setText(null);  
        }  
        super.updateItem(item, empty);  
    }  
}
```

# Working with Views

---

- ▶ Input for views is an ObservableList

```
public class CurrencyCell extends ListCell<Currency> {  
    @Override  
    protected void updateItem(Currency item, boolean empty) {  
        if( item != null && ! empty ) {  
            setText(item.getName());  
        } else {  
            setText(null);  
        }  
        super.updateItem(item, empty);  
    }  
}
```

# Working with Views

---

- ▶ Input for views is an ObservableList
- ▶ ListCell can be subclass and updateItem is called when a new item is associated with the Cell (can happen at ANY time!)

```
public class CurrencyCell extends ListCell<Currency> {  
    @Override  
    protected void updateItem(Currency item, boolean empty) {  
        if( item != null && ! empty ) {  
            setText(item.getName());  
        } else {  
            setText(null);  
        }  
        super.updateItem(item, empty);  
    }  
}
```

# Lab Views

- Setup the ListView

# Lab Views

---



# Lab Views

---

- ▶ Create a lib-Dir and copy there all jars from the fxgraph-libraries

# Lab Views

---

- ▶ Create a lib-Dir and copy there all jars from the fxgraph-libraries
- ▶ Open the CurrencyController

# Lab Views

---

- ▶ Create a lib-Dir and copy there all jars from the fxgraph-libraries
- ▶ Open the CurrencyController
  - ▶ make the ListView hold items of type Currency

# Lab Views

---

- ▶ Create a lib-Dir and copy there all jars from the fxgraph-libraries
- ▶ Open the CurrencyController
  - ▶ make the ListView hold items of type Currency
  - ▶ make the controller implement Initializable

# Lab Views

---

- ▶ Create a lib-Dir and copy there all jars from the fxgraph-libraries
- ▶ Open the CurrencyController
  - ▶ make the ListView hold items of type Currency
  - ▶ make the controller implement Initializable
- ▶ Add a subclass of ListCell named CurrencyCell as an inner-static-class

# Lab Views

---

- ▶ Create a lib-Dir and copy there all jars from the fxgraph-libraries
- ▶ Open the CurrencyController
  - ▶ make the ListView hold items of type Currency
  - ▶ make the controller implement Initializable
- ▶ Add a subclass of ListCell named CurrencyCell as an inner-static-class
- ▶ In the initialize-method setup the cellFactory

# Eclipse Databinding

# Eclipse Databinding

---



# Eclipse Databinding

---

- ▶ Eclipse Databinding is Domain-Model-Type agnostic

# Eclipse Databinding

---

- ▶ Eclipse Databinding is Domain-Model-Type agnostic
  - ▶ Abstract representation of a property

# Eclipse Databinding

---

- ▶ Eclipse Databinding is Domain-Model-Type agnostic
  - ▶ Abstract representation of a property
    - ▶ single value: `IValueProperty`

# Eclipse Databinding

---

- ▶ Eclipse Databinding is Domain-Model-Type agnostic
  - ▶ Abstract representation of a property
    - ▶ single value: `IValueProperty`
    - ▶ list value: `IListValueProperty`

# Eclipse Databinding

---

- ▶ Eclipse Databinding is Domain-Model-Type agnostic
  - ▶ Abstract representation of a property
    - ▶ single value: `IValueProperty`
    - ▶ list value: `IListValueProperty`
  - ▶ Representation of the property instance

# Eclipse Databinding

---

- ▶ Eclipse Databinding is Domain-Model-Type agnostic
  - ▶ Abstract representation of a property
    - ▶ single value: `IValueProperty`
    - ▶ list value: `IListValueProperty`
  - ▶ Representation of the property instance
    - ▶ single value: `IObservableValue`

# Eclipse Databinding

---

- ▶ Eclipse Databinding is Domain-Model-Type agnostic
  - ▶ Abstract representation of a property
    - ▶ single value: `IValueProperty`
    - ▶ list value: `IListValueProperty`
  - ▶ Representation of the property instance
    - ▶ single value: `IObservableValue`
    - ▶ list value: `IObservableList`

# Eclipse Databinding

---

- ▶ Eclipse Databinding is Domain-Model-Type agnostic
  - ▶ Abstract representation of a property
    - ▶ single value: `IValueProperty`
    - ▶ list value: `IListValueProperty`
  - ▶ Representation of the property instance
    - ▶ single value: `IObservableValue`
    - ▶ list value: `IObservableList`
- ▶ 2 instance can be synced through the `DatabindingContext`



# Eclipse Databinding

---

# Eclipse Databinding

---

- ▶ Creation of `IValueProperty` instances is done through Factories

# Eclipse Databinding

---

- ▶ Creation of `IValueProperty` instances is done through Factories
  - ▶ `JavaBeanProperties`, `EMFProperties`  
e.g. `EMFProperties.value(MyfondPackage.Literals.CURRENCY__NAME);`

# Eclipse Databinding

---

- ▶ Creation of `IValueProperty` instances is done through Factories
  - ▶ `JavaBeanProperties`, `EMFProperties`  
e.g. `EMFProperties.value(MyfondPackage.Literals.CURRENCY__NAME);`
  - ▶ `JFXUIProperty` for properties of JavaFX-Controls  
e.g. `JFXUIProperties.text()`

# Eclipse Databinding

---

- ▶ Creation of `IValueProperty` instances is done through Factories
  - ▶ `JavaBeanProperties`, `EMFProperties`  
e.g. `EMFProperties.value(MyfondPackage.Literals.CURRENCY__NAME);`
  - ▶ `JFXUIProperty` for properties of JavaFX-Controls  
e.g. `JFXUIProperties.text()`
- ▶ Creation of `IObservableValue`

# Eclipse Databinding

---

- ▶ Creation of `IValueProperty` instances is done through Factories
  - ▶ `JavaBeanProperties`, `EMFProperties`  
e.g. `EMFProperties.value(MyfondPackage.Literals.CURRENCY__NAME);`
  - ▶ `JFXUIProperty` for properties of `JavaFX-Controls`  
e.g. `JFXUIProperties.text()`
- ▶ Creation of `IObservableValue`
  - ▶ simple: `IValueProperty#observe(Object)`

# Eclipse Databinding

---

- ▶ Creation of `IValueProperty` instances is done through Factories
  - ▶ `JavaBeanProperties`, `EMFProperties`  
e.g. `EMFProperties.value(MyfondPackage.Literals.CURRENCY__NAME);`
  - ▶ `JFXUIProperty` for properties of `JavaFX-Controls`  
e.g. `JFXUIProperties.text()`
- ▶ Creation of `IObservableValue`
  - ▶ `simple: IValueProperty#observe(Object)`
  - ▶ `master-detail: IValueProperty#observeDetail(IObservableValue)`

# Lab DB

- ▶ Bind TextFields
- ▶ Update based on selection
- ▶ Change ListView to keep up-to-date



# Lab Eclipse DB

---

- ▶ In the Main#start call `JFXRealm.createDefault()`
- ▶ In `CurrencyController` create and initialize a field of type `WritableValue`
- ▶ In the `initialize-method`
  - ▶ Create an instance of `EMFDatabindingContext`
  - ▶ Create an instance `IValueProperty` for `CURRENCY__NAME` - through `EMFProperties`, `MyfondPackage.Literals`
  - ▶ Create an instance `IValueProperty` for `TextField#text` property through `JFXUIProperties`
  - ▶ Create an observable of the name `IValueProperty#observeDetail`
  - ▶ Create an observable of the text `IValueProperty#observe`

# Lab Eclipse DB

---

- ▶ Repeat the steps for the CURRENCY\_\_SYMBOL
- ▶ add an InvalidationListener to the currencyList's selectionModel and when call update master using IObservableValue#setValue
- ▶ Notice when running: ListCell is not updated!!!
- ▶ Create an IValueProperty for CURRENCY\_\_NAME
- ▶ Replace the list-setup through ListUtil.setupList(ListView,IValueProperty)

# Deployment

---

- ▶ The optimal way to deploy JavaFX applications is
  - ▶ Through the native install format (setup.exe, dmg, rpm, deb)
  - ▶ The JRE included so that no prerequisites are needed (e.g. Mac App Store requirement)
- ▶ JavaFX provides packageing tasks
  - ▶ Can be call on command line
  - ▶ Ant integration
- ▶ e(fx)clipse has a special file to configure the export named .fxbuild

# Lab Deploy

- Generate a native installer

# Lab Deployment

---

- ▶ Open the `build.fxbuild-File`
  - ▶ Enter infos into:
    - ▶ Vendor name: MY COMPANY
    - ▶ Application title: My App
    - ▶ Application version: 1.0.0
    - ▶ Application class: `application.Main`
    - ▶ Toolkit Type: `fx`
    - ▶ Packaging Format: `all`
- ▶ Click on „`ant build.xml` and run“

# FX & OSGi

# FX & OSGi

---

- ▶ JavaFX and OSGi are not natural friends
  - ▶ JavaFX is not JSRed hence it's in none of the OSGi-EEs
  - ▶ JavaFX is part of the JDK7 but not on a classpath
  - ▶ JavaFX is on the extension classpath in JDK8 but Equinox by default skips the extension classpath
- ▶ Most APIs have been adjusted to be OSGi-friendly (e.g. FXMLLoader takes a classloader)
- ▶ e(fx)clipse solves the integration problem for JDK7/8 in Kepler with a Adaptor Hook
  - ▶ Fragment to the system.bundle (org.eclipse.fx.osgi)
  - ▶ Fake bundle with JavaFX-packages (org.eclipse.fx.javafx)

# Lab FX & OSGi

- ▶ Create an FX-OSGi project
- ▶ Load an FXML-File



# Lab FX & OSGi

---

- ▶ Create a new project using File > New Project ... > OSGi Application Project
- ▶ Enter the following data on page 1
  - ▶ Bundle-ID-Prefix: `osgi.sample`
  - ▶ Execution Environment: JavaSE-1.8
- ▶ On the next page enter:
  - ▶ Product Name: MyOSGiApp
  - ▶ Eclipse DI: checked

# FX + OSGi + e4

---