MSc/ICY Software Workshop
Graphical User Interfaces

Manfred Kerber www.cs.bham.ac.uk/~mmk

Tuesday 10 December 2019

Overview

1. Pocket calculator computations, base types, simple strings, variables, static methods, JavaDoc
   Wed/Thu/Fri: 1st Lab Lecture (login, editor, javac, javadoc)
2. Classes, objects, methods, JUnit tests
   Wed/Thu/Fri: 2nd Lab Lecture (Eclipse)
3. Conditionals, 'for' Loops, arrays, ArrayList
4. Exceptions, I/O (Input/Output)
5. Functions, interfaces
6. Sub-classes, inheritance, abstract classes
7. Inheritance (Cont'd), packages
8. Graphics
9. ~~Revision~~
10. ~~Graphical User Interfaces~~
11. **Graphical User Interfaces** ~~**(Cont'd)**~~

Changes possible

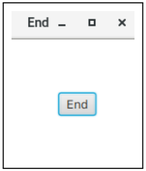
Model – View – Control

- When creating a graphical user interface (GUI) for a program, we try to keep the program functionality separate from the display elements (views) and interaction elements (controls). Sometimes (in case of big programs) we build a wrapper around the core program and build a so-called model through which we access the program code.
- For this reason, the approach of separation is often called Model-View-Controller approach, or MVC for short.

Different Approaches to GUIs in JavaFX

1. Write the Model, the View, and the Controller in the SAME CLASS. The Controller is written as an inner class.
2. Write the Model, the View, and the Controller in the SAME CLASS. The Controller is written as a function.
3. Write the Model and the Controller in DIFFERENT CLASSES with the Controller being a separate class.
4. Write the Model and the Controller in DIFFERENT CLASSES and the View as an fxml file.

Example – one button to click with a mouse:



Approach 1: Inner Class

```
public void start(Stage stage) throws Exception {
    Button endButton = new Button("End");
    EventHandler<MouseEvent> eventHandlerEnd =
        new EventHandler<MouseEvent>() {
            @Override
            public void handle(MouseEvent e) {
                System.out.println("This is the end.");
                System.exit(1);
            }
        };
    endButton.addEventFilter(MouseEvent.MOUSE_CLICKED,
                             eventHandlerEnd);
    Group root = new Group(endButton);
    Scene scene = new Scene(root, 150, 150);
    stage.setTitle("End");
    stage.setScene(scene);
    stage.show();
}
```

Approach 2: Function

```
private final EventHandler<MouseEvent> eventHandlerEnd =
    e -> {System.out.println("This is the end.");
          System.exit(1);
         };

public void start(Stage stage) throws Exception {
    Button endButton = new Button("End");
    endButton.addEventFilter(MouseEvent.MOUSE_CLICKED,
                             eventHandlerEnd);
    Group root = new Group(endButton);
    Scene scene = new Scene(root, 150, 150);
    stage.setTitle("End");
    stage.setScene(scene);
    stage.show();
}
```

Approach 3: Separate Classes

```
public class Main extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        Button endButton = new Button("End");
        EndingListener eventHandlerEnd =
            new EndingListener();
        endButton.addEventFilter(MouseEvent.MOUSE_CLICKED,
                                 eventHandlerEnd);
        Group root = new Group(endButton);
        Scene scene = new Scene(root, 150, 150);
        stage.setTitle("End");
        stage.setScene(scene);
        stage.show();
    }
}
```

Approach 3: Separate Classes (Cont'd)

```
public class EndingListener implements
                      EventHandler<MouseEvent> {
    public void handle(MouseEvent e) {
        System.out.println("This is the end.");
        System.exit(1);
    }
}
```

Approach 4: Classes plus fxml – Model

```
public class Main extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        Controller endControl = new Controller();
        stage.setScene(new Scene(endControl));
        stage.setTitle("End Control");
        stage.setWidth(300);
        stage.setHeight(200);
        stage.show();
    }
}
```

Approach 4: Classes plus fxml – Control

```
public class Controller extends VBox {
  public Controller() {
     FXMLLoader fxmlLoader =
       new FXMLLoader(getClass().getResource("end.fxml"));
     fxmlLoader.setRoot(this);
     fxmlLoader.setController(this);
     try {
         fxmlLoader.load();
     } catch (IOException exception) {
         throw new RuntimeException(exception);
     }
 }
 @FXML
 public void endApplication() {
     System.out.println("This is the end.");
     System.exit(1);
 }
}
```

Approach 4: Classes plus fxml – View

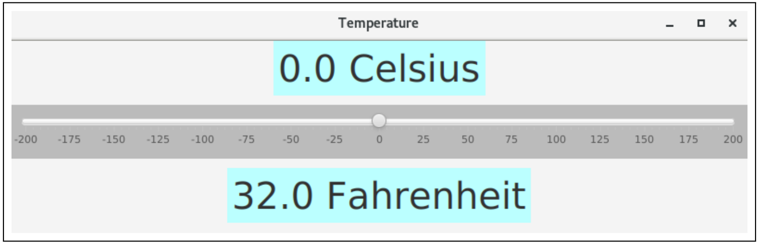
The view is provided in an xml-file

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<fx:root alignment="center"
         type="javafx.scene.layout.VBox"
         xmlns:fx="http://javafx.com/fxml">
  <Button text="End" onAction="#endApplication"/>
</fx:root>
```

Note, the application can be re-run with a changed fxml file without recompiling it.

The fxml-Approach to Temperature Conversion

Next we will look at a second example for the fxml approach, with the main part being a slider to convert temperatures from Celsius to Fahrenheit



Temperature Conversion — fxml – Main

```
public class TemperatureMain extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        Controller control = new Controller();
        stage.setScene(new Scene(control));
        stage.setTitle("Temperature");
        stage.setWidth(800);
        stage.setHeight(250);
        stage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

Temperature Conversion — fxml – Model

```
public class Temperature {
    private double celsius;
    private double fahrenheit;
    public Temperature(double celsius) {
        this.celsius = celsius;
        this.fahrenheit = 1.8 * celsius + 32.0;
    }
    public double getCelsius() {
        return celsius;
    }
    public void setCelsius(double celsius){
        this.celsius = celsius;
        this.fahrenheit = 1.8 * celsius + 32.0;
     }
    ...
}
```

Temperature Conversion — fxml – Controller

```
public class Controller extends VBox {
  @FXML private Label celsiusField;
  @FXML private Label fahrenheitField;
  @FXML private Slider temperatureSlider;
  private Temperature temperature = new Temperature(0);
  public Controller() {
    FXMLLoader fxmlLoader =
      new FXMLLoader(getClass().getResource("ui.fxml"));
      fxmlLoader.setRoot(this);
      fxmlLoader.setController(this);
      try {
        fxmlLoader.load();
        celsiusField.setText(Math.floor(10 *
          temperature.getCelsius())/10 + " Celsius");
        fahrenheitField.setText(Math.floor(10 *
          temperature.getFahrenheit())/10 + " Fahrenheit");
```

Temperature Conversion — fxml – Controller (Cont'd)

```
        temperatureSlider.valueProperty().addListener((
                   observable,oldValue, newValue) ->
        {temperature.setCelsius(temperatureSlider.getValue());
         celsiusApplication();
         fahrenheitApplication();
        });
      } catch (IOException exception) {
            throw new RuntimeException(exception);
      }
    }
    public void fahrenheitApplication() {
      double value = Math.floor(10 * temperature.getFahrenheit())/10
      fahrenheitField.setText(value + " Fahrenheit");
    }
    public void celsiusApplication() {
      double value = Math.floor(10 * temperature.getCelsius())/10;
      celsiusField.setText(value + " Celsius");
    }
    public void sliderApplication() {
      temperature.setCelsius(temperatureSlider.getValue());
    }
    }
```

Temperature Conversion — fxml – View as fxml-file

```
<?xml version="1.0" encoding="UTF-8"?>
<?import ...   ?>
<fx:root type="javafx.scene.layout.VBox" xmlns:fx="http://javafx
  <VBox spacing="10" alignment="center">
    <Label fx:id="celsiusField">
      <font>
        <Font name="verdana" size="40.0"/>
      </font>
    </Label>
    <Slider fx:id="temperatureSlider"
            max="200.0"
            min="-200.0"
            minorTickCount="5"
            showTickLabels="true"
            showTickMarks="true"
            value="0.0">
    </Slider>
    <Label fx:id="fahrenheitField">
      <font>
        <Font name="verdana" size="40.0"/>
      </font>
    </Label>
  </VBox>
  <stylesheets>
    <URL value="@temperatureConverter.css" />
  </stylesheets>
</fx:root>
```

Temperature Conversion — fxml – View refined by css-file

```
/* We give the celsius Field a background colour and a padding *

#celsiusField {
    -fx-background-color: #BBFFFF;
    -fx-padding: 5px;
}

/* We give the fahrenheit Field a background colour and a paddin
#fahrenheitField {
    -fx-background-color: #BBFFFF;
    -fx-padding: 5px;
}

/* The slider gets a background colour, a border width, and a pa
#temperatureSlider {
    -fx-background-color: #BBBBBB;
    -fx-border-width: 1px 1px 1px 1px;
    -fx-padding: 10px;
}
```

Recipe for Writing a GUI

We assume that the core program is already written. In order to write a GUI the following steps are taken:

1. Possibly create a model class (wraps underlying program objects, abstracts the program functionality).
2. Create a main class by writing the start method and the main method.
3. Create a controller class.
4. Create a view using an fxml file.
5. Possibly refine the view using a css file (cascading style sheet).

Getting Organized

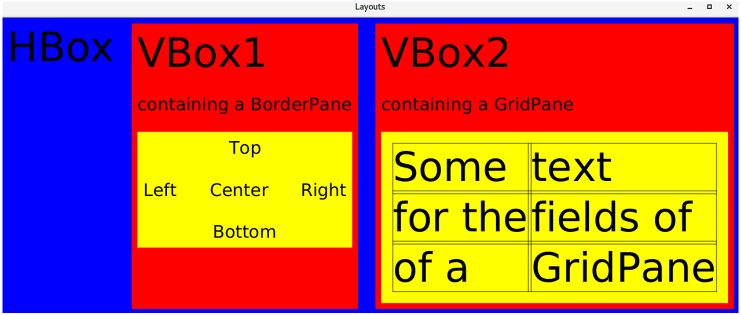
- The key to writing a good GUI is to get organized.
- Sketch how you want it to look on paper.
- Then make a list of all the classes, and of all the fxml and css files you will need.
- Write them one by one, working your way through the list.

Note on Separation

- The separation in Model – View – Control is clean.
- However, it is not necessarily very quick to write.
- Developers sometimes do not use it – instead they may write, for instance, the controller as either an inner class or as a function.
- This is less labour intense – but potentially messier to write and harder to maintain.

Layouts

There are different Layouts such as
HBox, VBox, BorderPane, and GridPane.



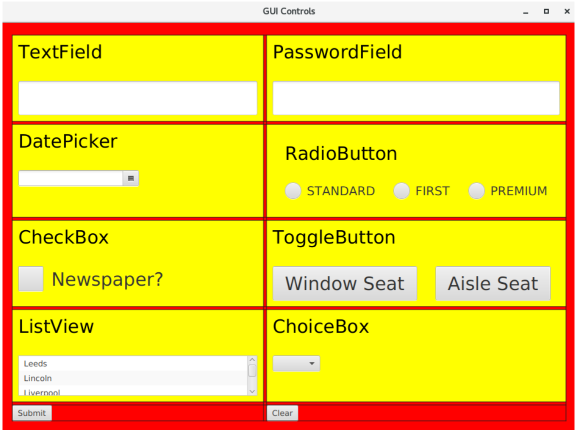
Layouts (Cont'd)

- HBox: A horizontal box that allows to put elements next to each other – from left to right.
- VBox: A vertical box that allows to put elements underneath each other – top down.
- BorderPane: An arrangement with 5 elements: top, bottom, left, right, and center.
- GridPane: A table kind of arrangement with columns and rows.

Details are summarized in the file Layouts.java in the examples directory.

GUI Controls

There are different Controls. Button and Slider. There are also TextField, PasswordField, DatePicker, RadioButton, CheckBox, ToggleButton, ListView, ChoiceBox, and some more.



See Controls.java.

GUI Controls (Cont'd)

- TextField: Allows the user to enter text
- PasswordField: Allows the user to enter text that is not visible on the screen.
- DatePicker: Allows the user to select a date from a calendar.
- RadioButton: Allows the user to select one of several choices.
- CheckBox: Allows the user to make a true/false choice.
- ToggleButton: Allows the user to select one of several choices.
- ListView: Allows the user to select from a given list.
- ChoiceBox: Allows the user to choose in a drop down box.

Model-View-Control Approach

We have seen an application with a clear separation into three different parts of a GUI (Celsius-Fahrenheit converter):

- The Model, which mirrors those part of the underlying system for which we build the GUI.
- The View, which presents the graphical presentation of the data.
- The Controller, which allows the user to interact with the system.

For small systems, often no model is created and the view and the controller are implemented in the same class. For a big system this has serious drawbacks.

Some Examples

We will look at a few examples to present some things that we can do with the primitives.

- Memory: How can we display images and change the display by clicking on them?
- Chess: How can we move on a chessboard?
- Change of Scene: How can we move from one scene to another?