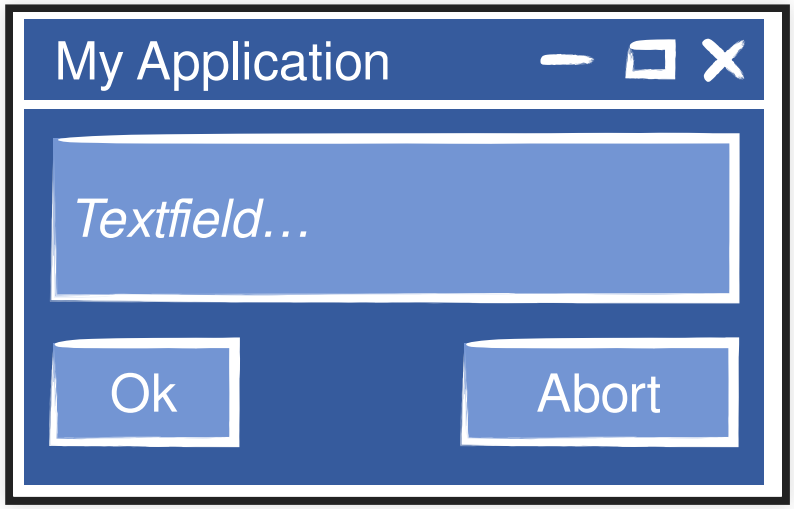
CHAPTER 9: GRAPHICAL USER INTERFACES

LEARNING OBJECTIVES

- Know the basic concepts (=classes) of JavaFX and their interaction
- Be able to use the different options for using windows in JavaFX
- Be able to use JavaFX controls for building user interfaces
- Know the JavaFX panes and be able to use them to layout a window
- Be able to display the layout of a window in the form of a layout hierarchy

9.1 OVERVIEW

Graphical user interfaces (GUIs) are nowadays created using so-called GUI libraries or GUI frameworks.



JavaFX (standard) and Swing (predecessor) are commonly used for Java, but the use of Swing is no longer encouraged, so we will discuss JavaFX here.

JAVAFX OFFERS

- Windows as rectangular containers for control elements
- UI elements (buttons, text fields, selection menus, etc.) - also called controls
- Event handling (e.g. what happens when the user clicks on a button?)

Not covered here:

- Layout design via CSS (similar to web development)
- Graphical primitive operations (e.g. drawing lines and points, setting colours)
- JavaFX FXML: XML-based language for GUI design

LAUNCHING A JAVAFX APPLICATION

A JavaFX application is not started with `main()`, but by a so-called launcher. This has to do with the life cycle of a JavaFX application, which is explained below. A JavaFX application is based on the Application base class, from which it is typically derived.

Class Application implements the following life cycle:

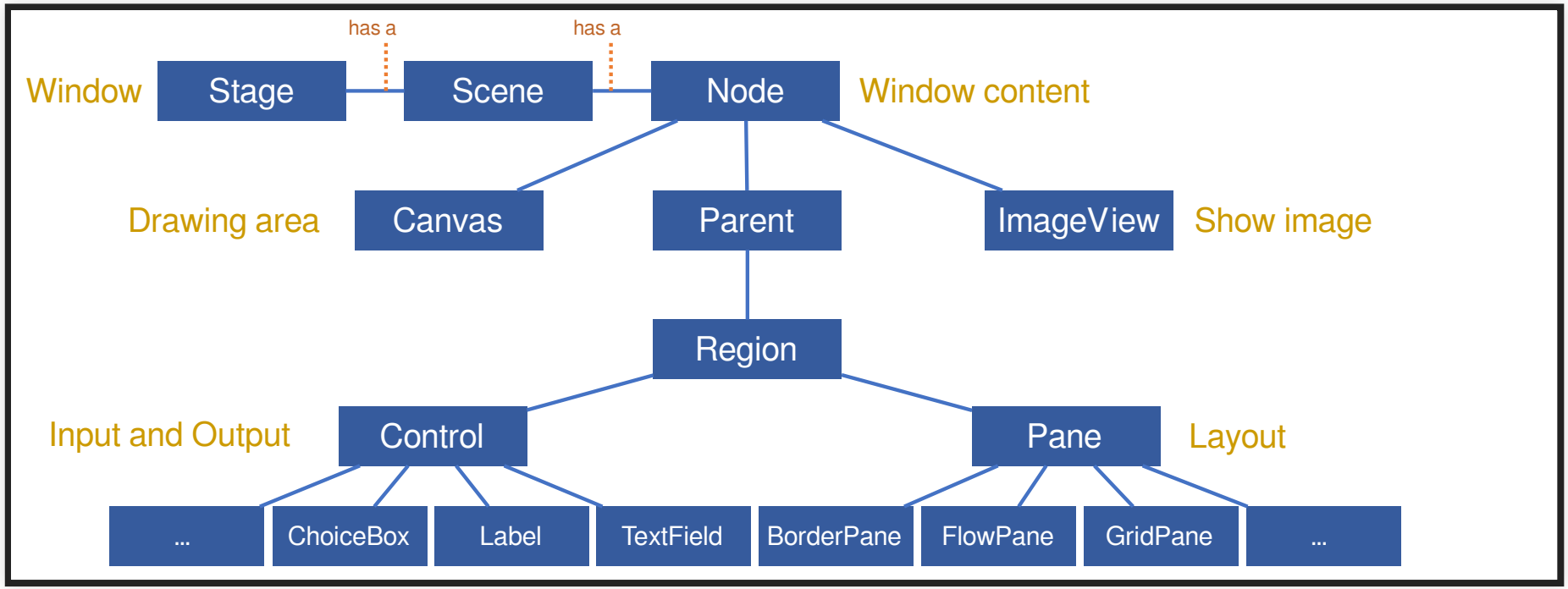
- when an application is started, an instance of `Application` is created, on which the `init` method is then called.
- `init` is already predefined and can be overwritten.
- The `start(Stage stage)` method is then called, passing a window (class `Stage`) in which the first scene graph (see below) is displayed. The `start` method should be overridden by any application. GUI controls can then be created and placed in the stage.
- When the application is terminated with `Platform.exit()`, the stop method is called before the Java process is terminated, which can be overwritten if necessary, e.g. to close open files.

A JavaFX application typically uses the following objects:

- Windows are represented in JavaFX by instances of the class `Stage` (see section 9.2 for more details).
- A `stage` contains a so-called scene graph, which is a tree of nodes. Each node element is typically a region object or a control object (there are other subclasses which we will not go into here due to time constraints).
- Region objects can be Panes, which are further subdivided into Border-, Flow- and GridPane. These are used for the layout of windows (see section 9.4).
- Control objects are the GUI controls, i.e. the UI elements within each window. These are explained in section 9.3.

The API documentation for JavaFX can be found at
https://docs.oracle.com/javafx/2/api/.

The following class diagram shows the classes derived from Node to illustrate the above. For simplicity, some "intermediate" classes are not shown.



EXERCISE: OVERVIEW JAVAFX (1)

Which statements are correct?

- ☐ Control objects are responsible for the layout and stage objects for the windows.
- ☐ A JavaFX application is started like any non-JavaFX programme.
- ☐ You can use life cycle methods to execute your own Java code before starting and when closing the application.

EXERCISE: OVERVIEW JAVAFX (2)

Which statements are correct?

- ☐ A stage has a scene and the scene has exactly one node.
- ☐ A stage has one or more scenes and each scene have at least one node.
- ☐ Control objects are responsible for the input and output, pane objects are responsible for the layout and stage objects for the windows.

9.2 WINDOWS

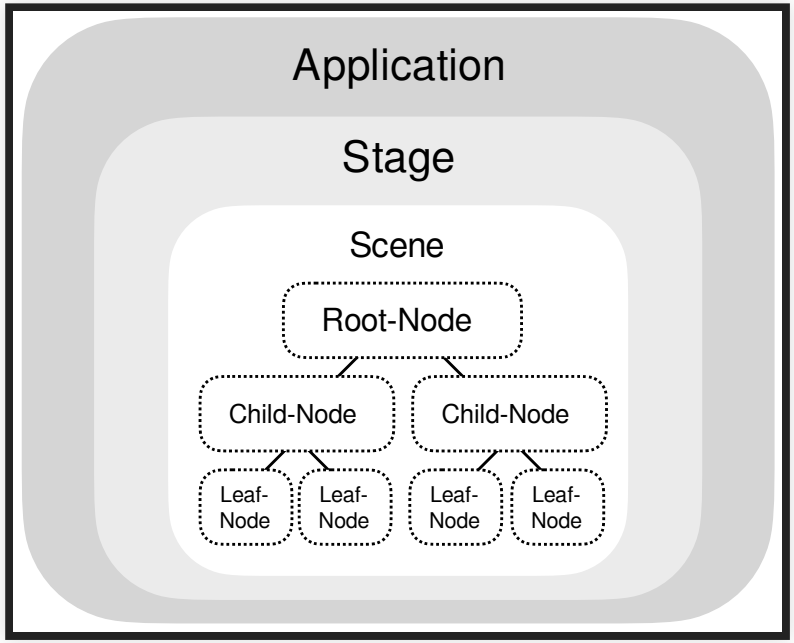
Windows are the basic elements of every GUI. In JavaFX, windows are realised by instances of the Stage class. Below is the classic "Hello World!" program with JavaFX:

```
public class HelloWorldMitJavaFX extends Application {
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("Hello World Window");
        Pane pane = new FlowPane();
        pane.getChildren().add(new Label("Hello World!"));
        Scene scene = new Scene(pane, 300, 200);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    static public void main(String[] args) {
        launch(args);
    }
}
```

Unzip javafx-helloworld.zip and import the folder as "Existing Maven Project".

The application is started by the command `launch` in `main` (see life cycle), whereupon a `HelloWordWithJavaFX` instance is created, on which `start` is called with a window as a parameter. The title of the window is then set and the window receives a first (and here only) scene with a label element in it, which displays the text "Hello World!". At the end, the `primaryStage` window is displayed. The three system buttons (Minimise, Maximise, Close) are displayed by default. The close button closes the application. The window can be resized (resizable).

Applications in JavaFX can be structured as shown in the example below.



The tree structure in the scene results in a layout hierarchy.

Several windows can also be displayed. To do this, simply create several 'Stage' instances. The example below shows two windows, the second of which is a modal window, i.e. it must be used first. The first window can only be used once the second window has been closed.

```
public void start(Stage primaryStage) throws Exception {
    primaryStage.setTitle("Window 1 (amodal)");
    Pane pane = new FlowPane();
    pane.getChildren().add(new Label("Hello, this is Window 1!"));
    Scene scene = new Scene(pane, 300, 200);
    primaryStage.setScene(scene);
    primaryStage.show();

    Stage fensterZwei = new Stage();
    Pane pane2 = new FlowPane();
    pane2.getChildren().add(new Label("Hello, this is window 2!"));
    pane2.setStyle("-fx-background-color: lightgrey");
    fensterZwei.setTitle("Window 2 (modal)");
    fensterZwei.setScene(new Scene(pane2, 300, 100));
    fensterZwei.initOwner(primaryStage);
    fensterZwei.initModality(Modality.WINDOW_MODAL); // Modal window
    fensterZwei.show();
}
```

Unzip javafx-two-windows.zip and import the folder as "Existing Maven Project".

EXERCISE: WINDOWS

Which statements are correct?

- ☐ If a modal window is open, other windows of the application cannot be used.
- ☐ Every JavaFX application has exactly one window.
- ☐ The second window in JavaFX must always be a modal window.
- ☐ A JavaFX application can have one or more windows.

9.3 HANDLING EVENTS

Each control can trigger certain events in response to a user interaction; for example, a button triggers an ActionEvent when it is pressed. In order to define a reaction to the button click, a so-called handler method must be registered with the button object itself. The method is then called by the JavaFX framework when the button is clicked. This also explains the difference between the framework and the library: the framework calls the "user code", the library is called by the "user code"; e.g. `double val = Math.sin(3.2);`

Event handling example

For the last example, you could, for example, specify that the name entered in the text field should be output to the console each time the button is clicked:_

```
okButton.setOnAction(new EventHandler<ActionEvent>() {
  public void handle(ActionEvent event) {
    System.out.println("Name: " + nameTextField.getText());
  }
});
```

An instance of a so-called anonymous class is passed to the `setOnAction` method. All we know about the anonymous class defined above is that it implements the `EventHandler` interface with the parameter ActionEvent. The `handle` method is the actual handler to be specified for the event. You could also pass instances of other classes and define them specifically for this purpose. However, the above method is preferable as it requires less code. With lambdas (see chapter 10), event handlers can be formulated even more briefly.

The last mentioned method using a separate class would look something like this:

The handler class requires a reference to the text field for the name in order to be able to output the name entered. You can see at a glance that this solution is very cumbersome and requires more code.

```
// Handler class with handle method for event handling
class MyHandler implements EventHandler<ActionEvent> {
  private TextField nameTextField;
  public MyHandler(TextField nameTextField) {
    this.nameTextField = nameTextField;
  }
  public void handle(ActionEvent event) {
    System.out.println("Name: " + nameTextField.getText());
  }
}
```

```
// Create and set the handler
okButton.setOnAction(new MyHandler(nameTextField));
```

Unzip javafx-events.zip and import the folder as "Existing Maven Project".

The last mentioned way via a lambda expression would look something like this (see chapter 10):

```
okButton.setOnAction(e ->
  System.out.println("Name: "+ nameTextField.getText()));
```

As the different controls trigger different events, the events and the corresponding event handlers are explained together with the controls.

EXERCISE: EVENT HANDLING

Which statements are correct?

- ☐ Event handling must always use lambda expressions.
- ☐ The event handling must provide a handling for every possible event.
- ☐ To handle a click event on a button, a handler must be registered with the setOnAction method.
- ☐ There are various ways of handling an event, but the treatment using lambda expressions is favoured here.

9.4 CONTROLS

The windows of a graphical user interface typically display information and usually offer options for changing the information displayed. JavaFX offers a number of predefined controls for this purpose.

All controls are derived from the Control class (see class model), which in turn inherits from Parent. Controls are displayed by adding them to the child list of a suitable parent object. Each parent object has a list of child objects (children). Suitable parent objects are all types of containers, such as the FlowPane from a previous example.

```
Pane container = new FlowPane();                    // create container
container.getC,hildren().add(new Label("Ein Text")); // add label
```

The following JavaFX controls are discussed:

Control	Description
Button	Action button to click on
CheckBox	Selection box (checked, not checked, undef.)
ChoiceBox and ComboBox	Dropdown list for a few or many elements
Label	Text field for displaying text
ListView	List of objects with single/multiple selection

Control	Description
PasswordField	Password input field with hidden input
RadioButton	Single selection from a group of options
TableView, TableColumn, TableCell	Table with columns (TableColumn) and cells (TableCell)
TextField and TextArea	Single-line or multi-line text input
ToogleButton	Button with state *selected/not selected*

The following example shows the use of some of the controls to be discussed:

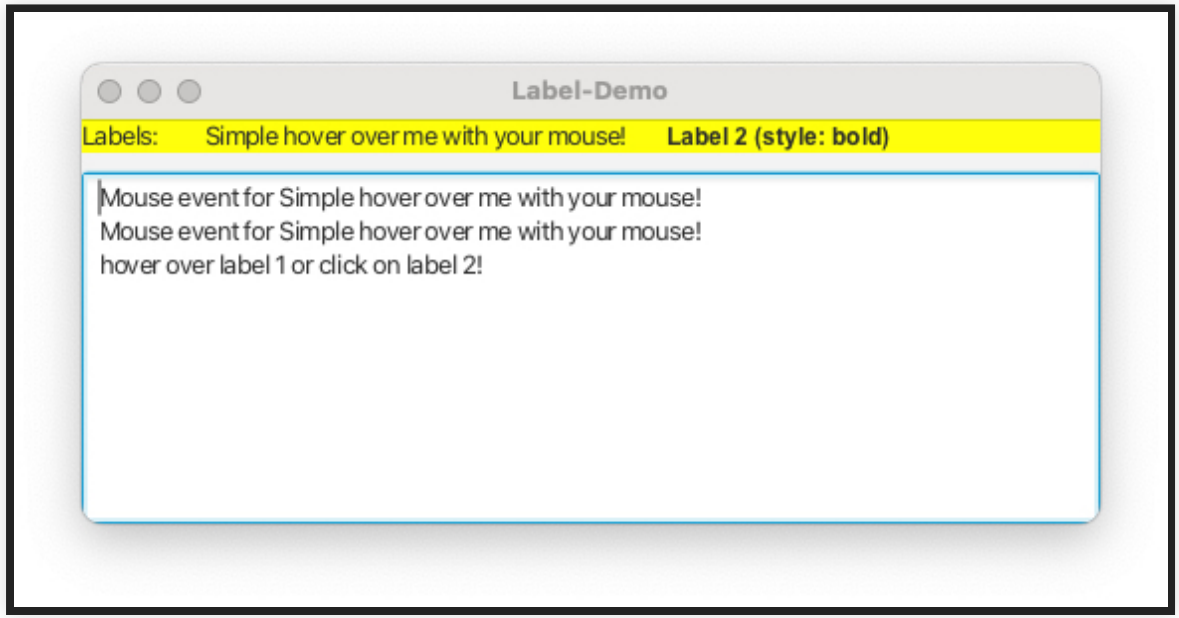
```
public class Example1 extends Application {
    public void start(Stage primaryStage) {
        TextField nameTextField = new TextField();
        Button okButton = new Button("OK");
        primaryStage.setTitle("Input your name");
        Pane pane = new FlowPane();
        pane.getChildren().add(new Label("Name: "));
        pane.getChildren().add(nameTextField);
        pane.getChildren().add(okButton);
        Scene scene = new Scene(pane, 300, 50);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    static public void main(String[] args) { launch(args); }
}
```

First, the three controls (text field, button, label) are created, then added to the FlowPane as children using the add command. If the button is pressed, *nothing* happens! We still need to define *event handling*!

Checkout the example:

Unzip javafx-example1.zip and import the folder as "Existing Maven Project".

LABELS


Label-Demo
Labels:
Simple hover over me with your mouse!
Label 2 (style: bold)
Mouse event for Simple hover over me with your mouse!
Mouse event for Simple hover over me with your mouse!
hover over label 1 or click on label 2!

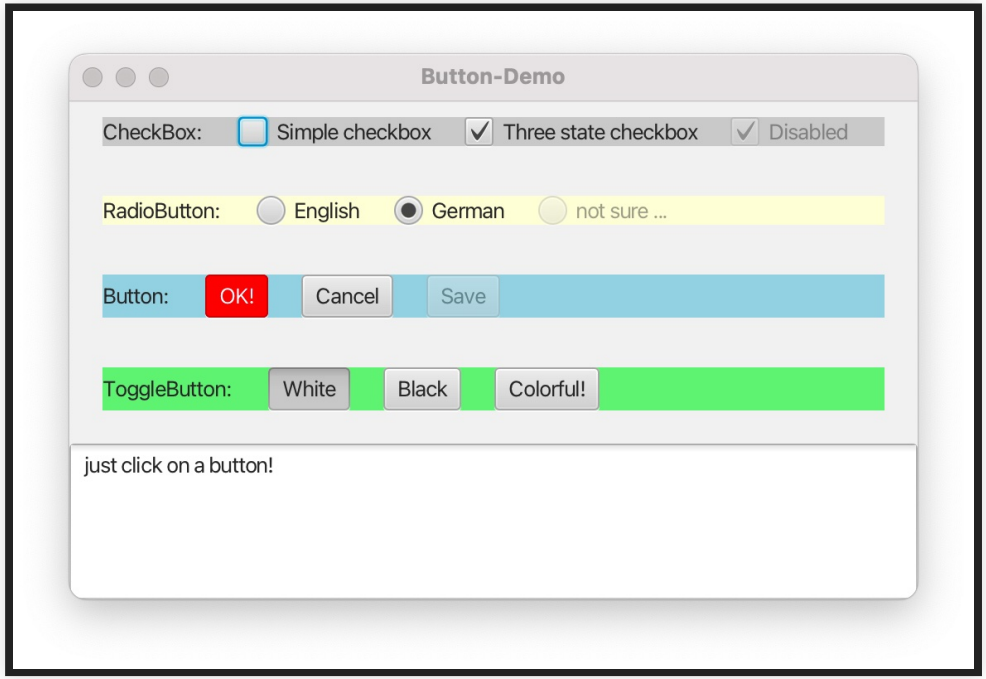
Example application for labels

Unzip javafx-labels.zip and import the folder as "Existing Maven Project".

A label is used to represent text that can be positioned and styled.

- String attribute text with getter/setter
- Constructor with string parameter to initialize the text
- Event handling (rare):
  - `setOnMouseEntered(EventHandler<MouseEvent>)` is called when the mouse pointer is hovered over the label
  - `setOnMouseClicked(EventHandler<MouseEvent>)` is called when a mouse click occurs over the label

BUTTON, CHECKBOX, RADIOBUTTON, TOGGLEBUTTON



Example application for Button, CheckBox, RadioButton, ToggleButton

Unzip javafx-buttons.zip and import the folder as "Existing Maven Project".

All buttons have certain important properties...

- Attribute `text`: Label of the button
  - Access via getters/setters
- Constructor: Passing button label as parameter
- Event handling
  - triggered event on click: ActionEvent
  - Setting the event handler:
    `setOnAction(EventHandler<ActionEvent>)`
- 1 out of N selection logic by forming a `ToggleGroup`: In addition to adding them to a container for display, radio or toggle buttons must be associated by a `ToggleGroup`. Use `setToggleGroup(ToggleGroup)` on each button to connect them

- Set the style attributes like with CSS (see websites) with `setStyle(<Style-String>)`, example see OK! button (applies to all controls and containers, see FlowPanes)
- Selected property for `CheckBox`, `RadioButton` and `ToggleButton`:
  - Getter: `boolean isSelected()`
  - Setter: `void setSelected(boolean)`
- Checkboxes can have three states: `checked`, `unchecked` (default) and as an option `undefined` (this must be explicitly allowed, see button demo).

TEXTFIELD, TEXTAREA, PASSWORDFIELD

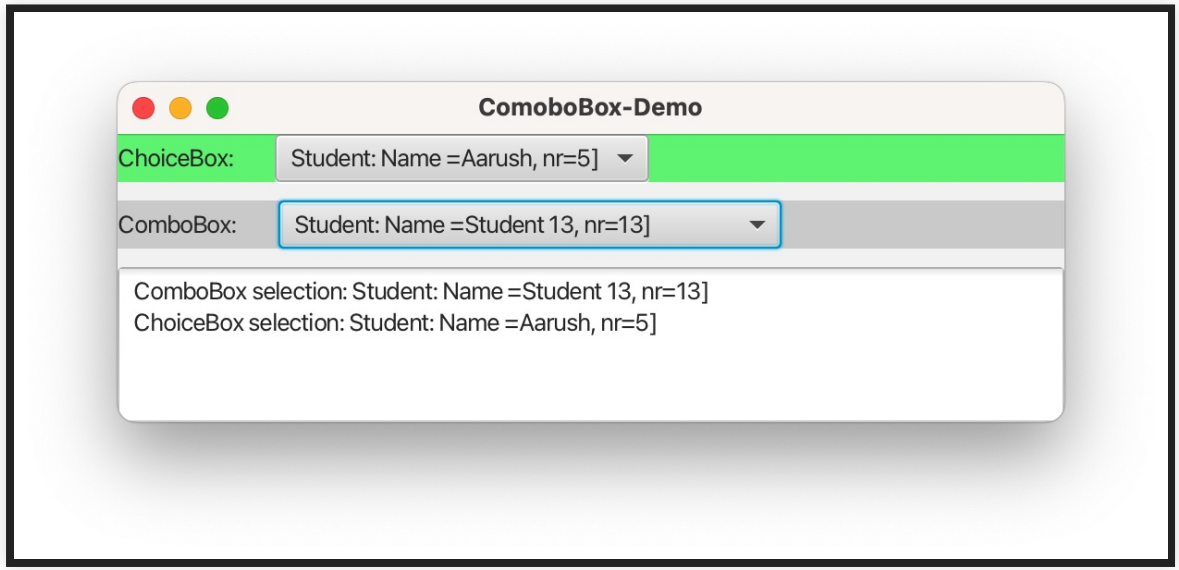
Example application for TextField, TextArea, PasswordField

Unzip javafx-textfields.zip and import the folder as "Existing Maven Project".

All three classes display text. TextArea can display multi-line text with automatic scrollbars. Here are some useful properties...

- String attribute "text" with getter/setter
- Constructor with string parameter for initialising the text (not for PasswordField)
- Event handling
  - `setOnMouseClicked(EventHandler<MouseEvent>)`: Mouse click in text field
  - `setOnAction(EventHandler<ActionEvent>)`: Enter key pressed in text field

CHOICEBOX, COMBOBOX



Example application for ChoiceBox, ComboBox

Unzip javafx-boxes.zip and import the folder as "Existing Maven Project".

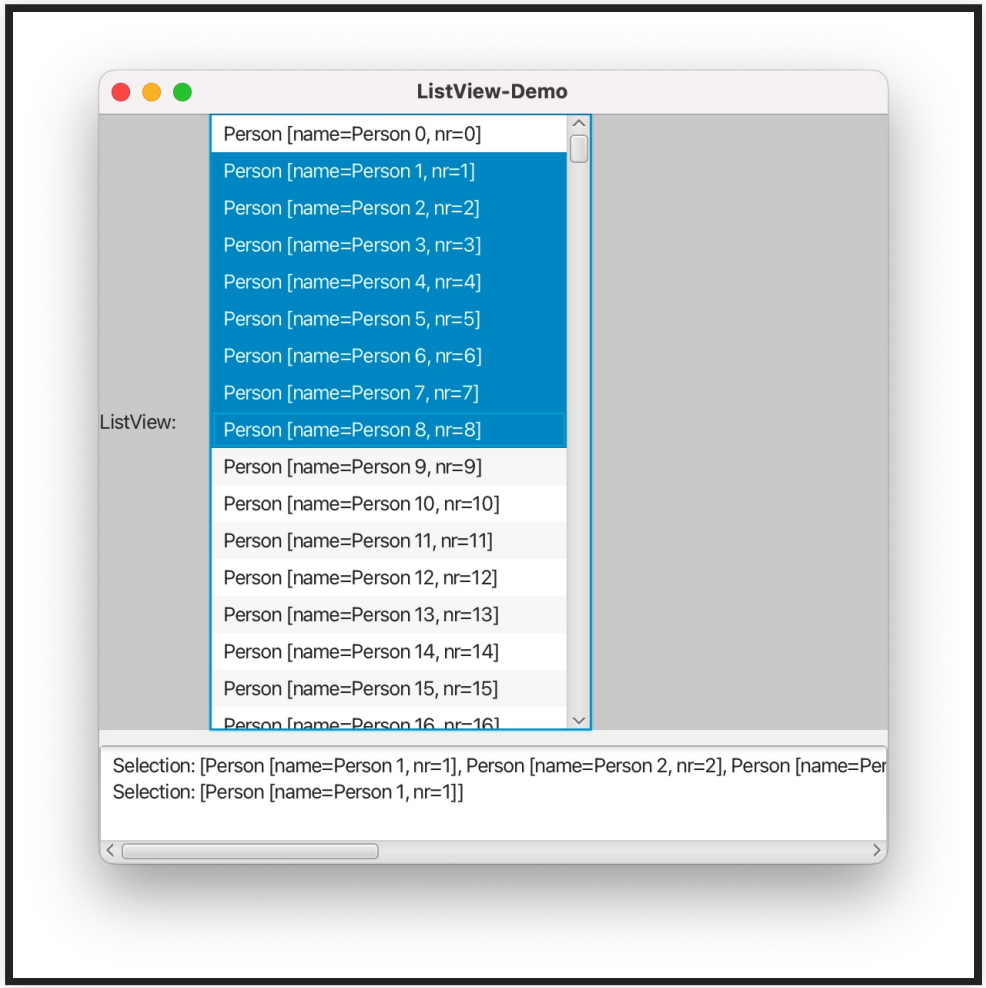
ComboBoxes, also known as dropdowns, are used to select an entry from a list. The list is expanded for the selection process, after the selection it collapses again and shows the selected entry. The ChoiceBox class is suitable for selecting from small lists, ComboBox for selecting from large lists.

Some useful properties:

- Attribute `ObservableList<list element type>` items with getter/setter; an example of the creation and filling of an `ObservableList` can be found in the ComboBoxes class.
- Constructor with optional parameter for "items"
- Event handling with `setOnAction(EventHandler<ActionEvent>)`: Entry selected
- Access to selected element:

```
ComboBox<String> cb = new ComboBox<>(observableList);
String selectedItem = cb.getSelectionModel().getSelectedItem();
```

LISTS WITH LISTVIEW



Example application for lists

Unzip javafx-listviews.zip and import the folder as "Existing Maven Project".

A `ListView<T>` represents an ObservableList of objects of a certain type `T` and enables the selection of one or more objects.

Useful properties:

- `ObservableList<T>` items: attribute for list elements with getter/setter
- Constructor with optional parameter for `items`
- Set multiple selection (default is single selection):

```
listView.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE)
```

- Event handling (if entry is selected):

```
listView.setOnMouseClicked(EventHandler<MouseEvent>)
```

- Reading the selected entries:

```
listView.getSelectionModel().getSelectedItems()
```

TABLES

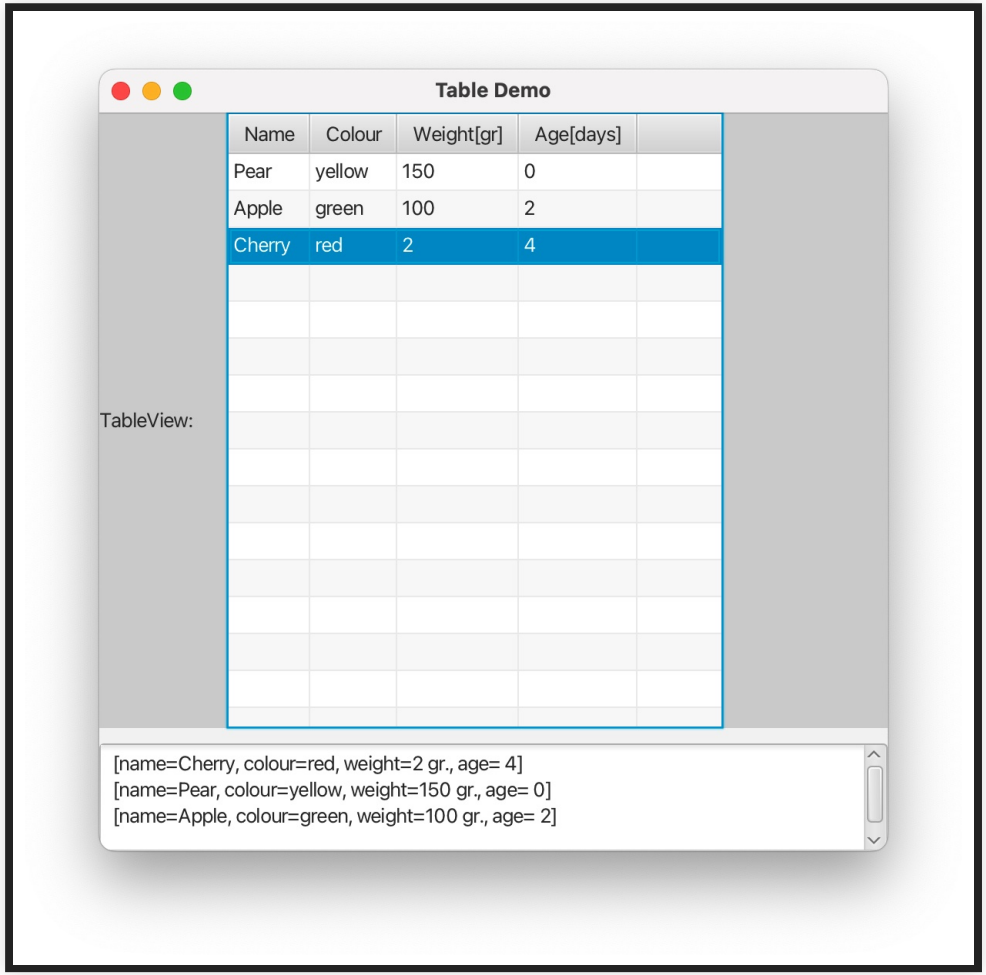


Table Demo
TableView:
Name
Colour
Weight[gr]
Age[days]
Pear
yellow
150
0
Apple
green
100
2
Cherry
red
2
4
[name=Cherry, colour=red, weight=2 gr., age= 4]
[name=Pear, colour=yellow, weight=150 gr., age= 0]
[name=Apple, colour=green, weight=100 gr., age= 2]

Example application for tables

Unzip javafx-tables.zip and import the folder as "Existing Maven Project".

Similar to ListView, a TableView represents an ObservableList of a certain type, whereby each object is displayed in a line.

Useful properties:

- ObservableList items: attribute for list elements with getter/setter
- Constructor with optional parameter for "items"
- Set multiple selection (default is single selection):

```
tableView.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE)
```

- Event handling (if entry is selected):

```
tableView.setOnMouseClicked(EventHandler<MouseEvent>)
```

- Reading the selected entries:

```
tableView.getSelectionModel().getSelectedItems()
```

Preparing the table columns

Before the data can become visible, a TableColumn instance must be initialised for each column. This is done by setting the column title and a CellValueFactory. In the example, instances of the PropertyValueFactory class are used to do this, which extracts the corresponding attribute value from the Row object via Java reflection using the attribute name (e.g. "name") by calling a getter (e.g. getName()).

Editable tables

If you want to be able to edit the cells of a table, you must first call `setEditable(true)`. Then, for each column whose cells are to be edited, a CellFactory must be set, which takes care of converting the cell data between the object attribute (e.g. age) and the text display in the cell ("age" column). It provides a converter that implements the StringConverter interface. In the example a standard converter is used for the string attributes and a separate one for the int attributes (weight, age).

9.5 LAYOUT

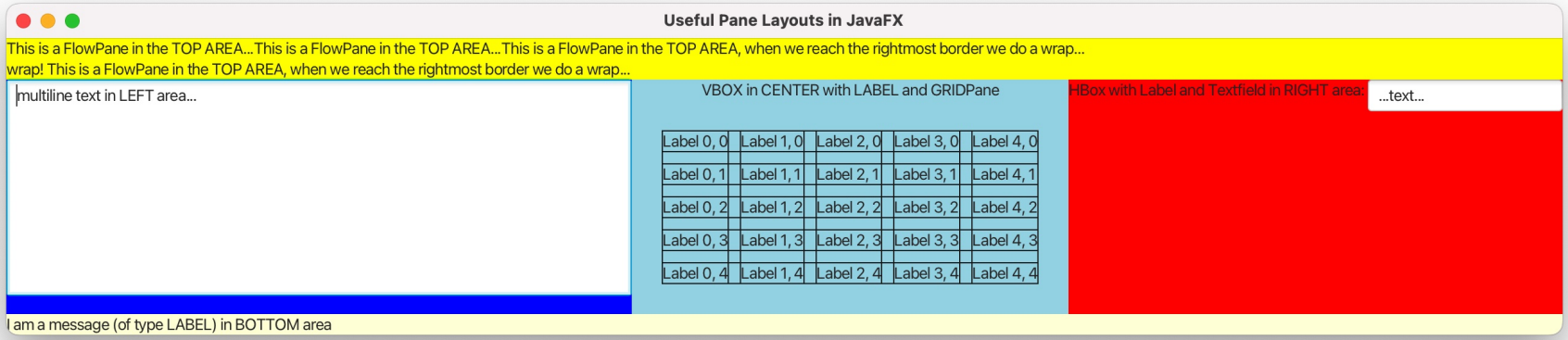
The explicit positioning of the controls of a window based on specific coordinates is rarely used in practice, mostly because the window should be resizable, controls should be shown or hidden or because the available space (screen resolution) changes depending on the target platform. In all these cases, JavaFX offers pane classes with predefined layout strategies.

Unzip javafx-layout.zip and import the folder as "Existing Maven Project".

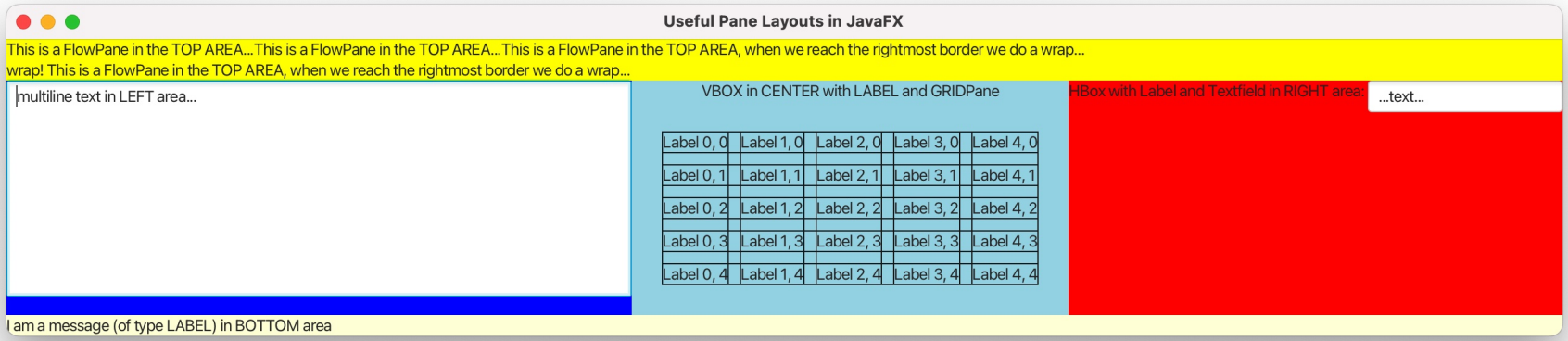
Five important Pane types are introduced below:

- `FlowPane`: Arrangement of the controls from left to right with wrapping at the right container border
- `BorderPane`: Division of the container rectangle into five areas: Top, Bottom, Right, Left, Center, each area can contain **exactly one** Parent object (see inheritance hierarchy)
- `GridPane`: Matrix-like structure with any number of rows and columns (depends on filling!); each cell of the matrix can contain an object of type Parent
- `HBox`: all controls are displayed in one row
- `VBox`: all controls are displayed in a column

Others (not discussed here): Anchor-, Tile-, Stackpane


Useful Pane Layouts in JavaFX
This is a FlowPane in the TOP AREA...This is a FlowPane in the TOP AREA...This is a FlowPane in the TOP AREA, when we reach the rightmost border we do a wrap...
wrap! This is a FlowPane in the TOP AREA, when we reach the rightmost border we do a wrap...
multiline text in LEFT area...
VBOX in CENTER with LABEL and GRIDPane
Label 0, 0
Label 1, 0
Label 2, 0
Label 3, 0
Label 4, 0
Label 0, 1
Label 1, 1
Label 2, 1
Label 3, 1
Label 4, 1
Label 0, 2
Label 1, 2
Label 2, 2
Label 3, 2
Label 4, 2
Label 0, 3
Label 1, 3
Label 2, 3
Label 3, 3
Label 4, 3
Label 0, 4
Label 1, 4
Label 2, 4
Label 3, 4
Label 4, 4
HBox with Label and Textfield in RIGHT area:
...text...
I am a message (of type LABEL) in BOTTOM area

FLOWPANE



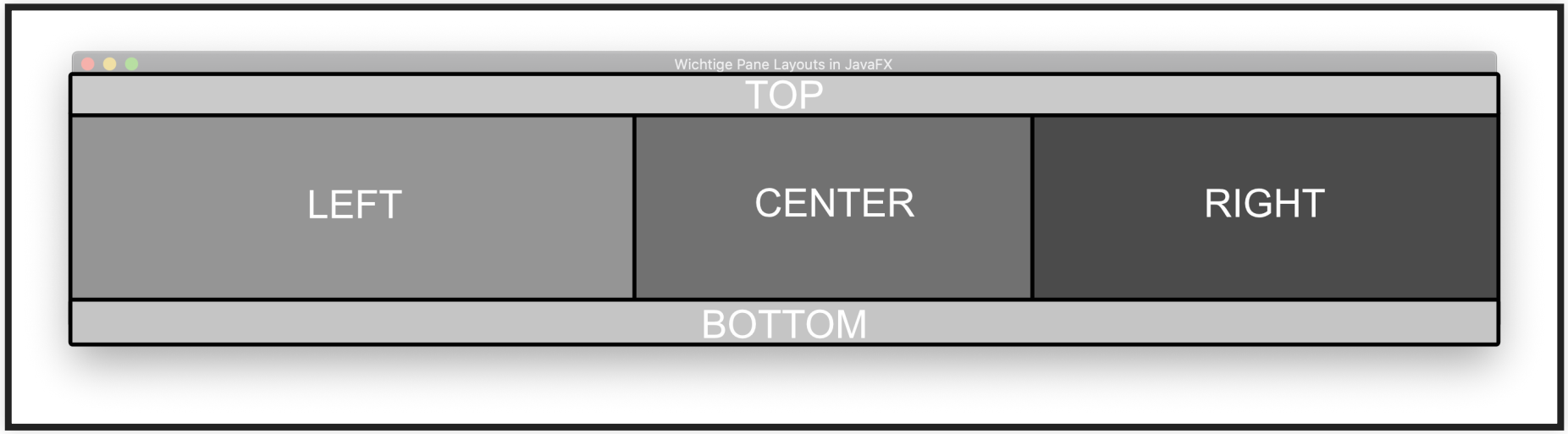
A FlowPane arranges the added elements from left to right. If the right pane border is reached, a new line is started with the next element.

Procedure for adding elements to a FlowPane:

Elements to be inserted (pane or control) are inserted using the `add` operation.

```
flowPane.getChildren().add(parent)
```

BORDERPANE



A BorderPane divides the container rectangle into the five areas shown: top, bottom, right, left and center.

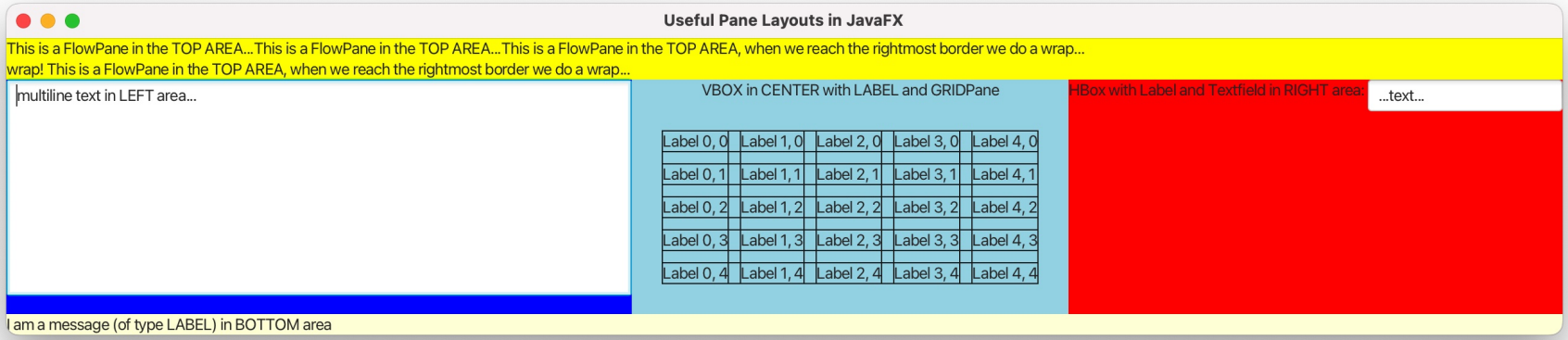
Procedure for adding elements to a BorderPane:

Panes or controls are inserted into the areas by using setter operations. There is a corresponding setter for each area, e.g. for center:

```
borderPane.setCenter(element)
```

Please be aware: Several controls (e.g. two buttons) must be packed into a pane, which is then placed in the BorderPane in the corresponding position.

VBOX AND HBOX



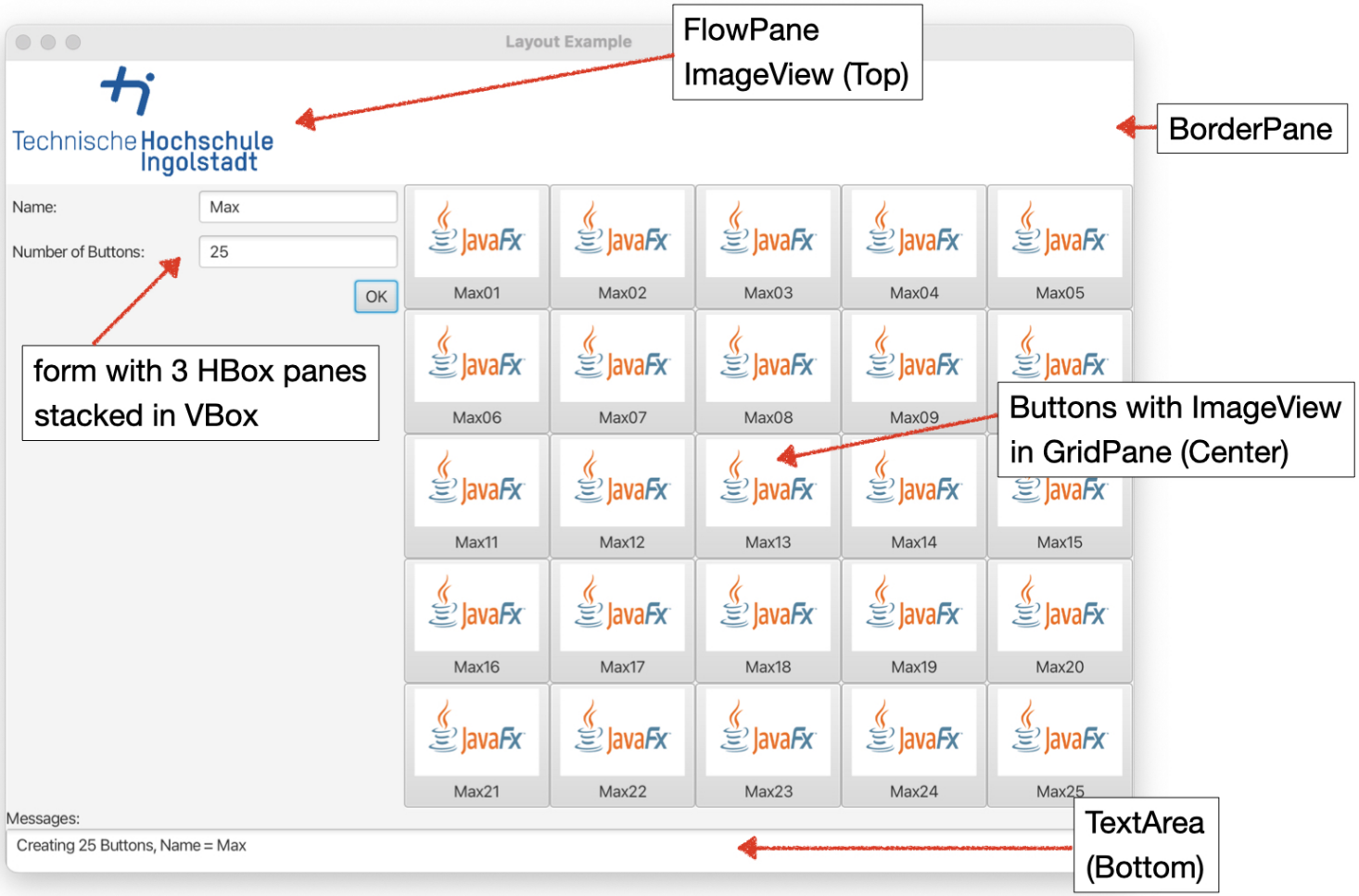
An `HBox` divides the container of the pane into one line, the width of which depends on the elements arranged from left to right and the spaces set (with `setHGap(double)`). Similarly, a `VBox` divides the pane container into a column, the height of which depends on the elements arranged from top to bottom and the gaps set (with `setVGap(double)`).

Procedure for adding elements:

Elements to be inserted (pane or control) are inserted using the `add` operation.

```
vbox.getChildren().add(parent) // same for hbox
```

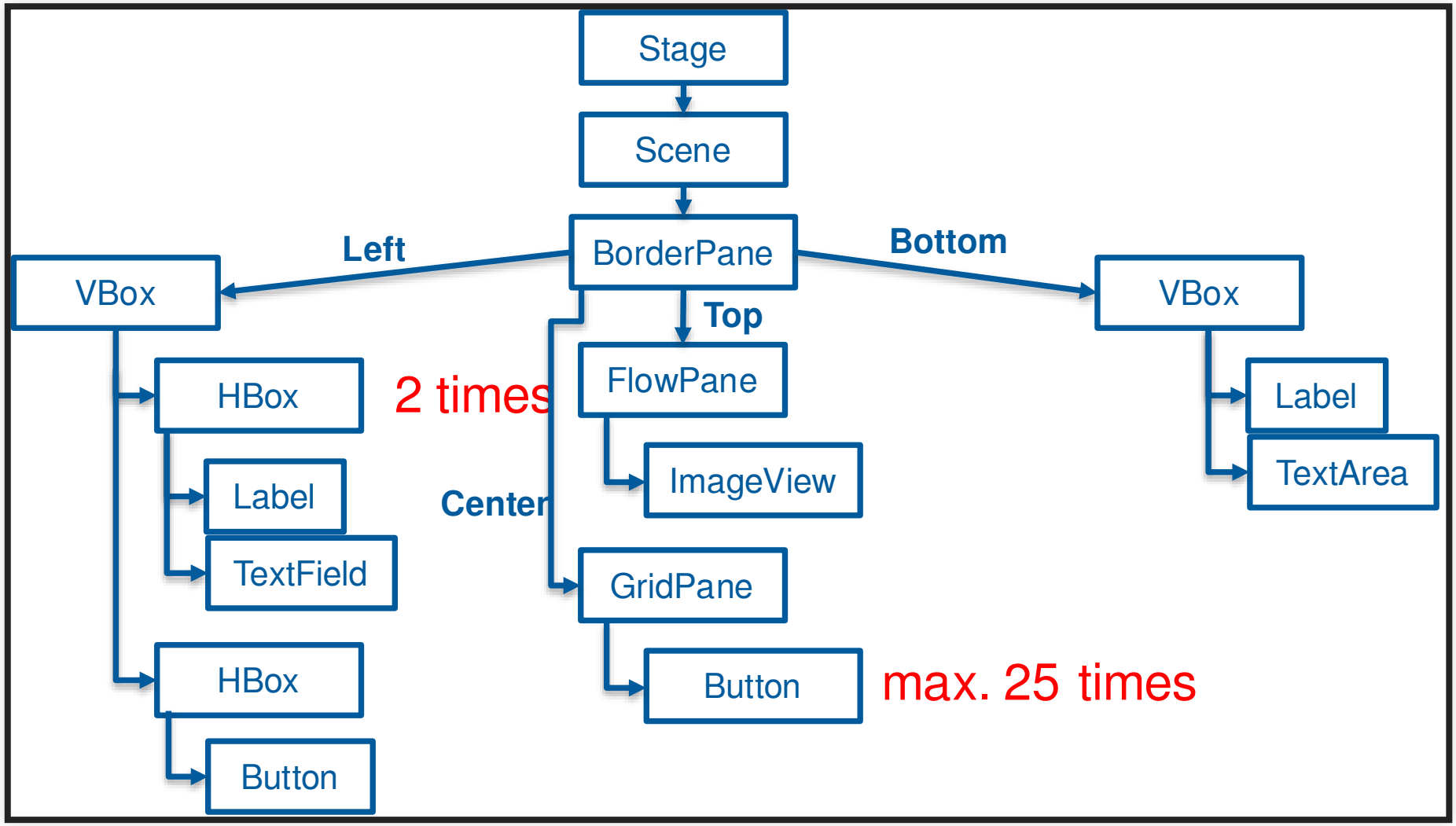
EXAMPLE OF LAYOUT AND CONTROL USAGE



Unzip javafx-layout-example.zip and import the folder as "Existing Maven Project".

LAYOUT HIERARCHY OF THE EXAMPLE

To make the layout easier to understand, a so-called layout hierarchy can be used as a tree structure. It represents the objects involved with their types and any layout information. It is obtained "bottom-up" from the source code or "top-down" as part of a design process, typically from paper sketches.


Stage
Scene
BorderPane
Left
Bottom
Top
Center
VBox
HBox
Label
TextField
HBox
Button
2 times
FlowPane
ImageView
GridPane
Button
max. 25 times
VBox
Label
TextArea

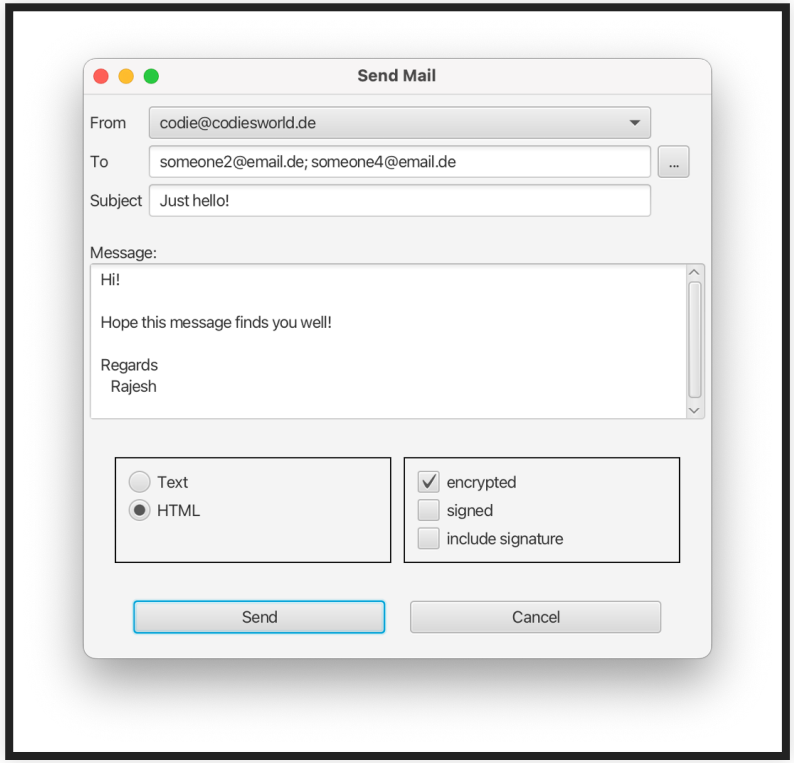
EXERCISE: LAYOUT

Which statements are correct?

- ☐ In the GridPane, all positions must be occupied by elements.
- ☐ In the BorderPane, some of the areas can also be left empty.
- ☐ The predefined panes each offer different layout strategies for their content.
- ☐ HBox and FlowPane offer an identical layout mechanism.
- ☐ Instead of using panels, you can also place the controls at the required x/y position in the window without any disadvantages.

9.6 JAVAFX EXAMPLE

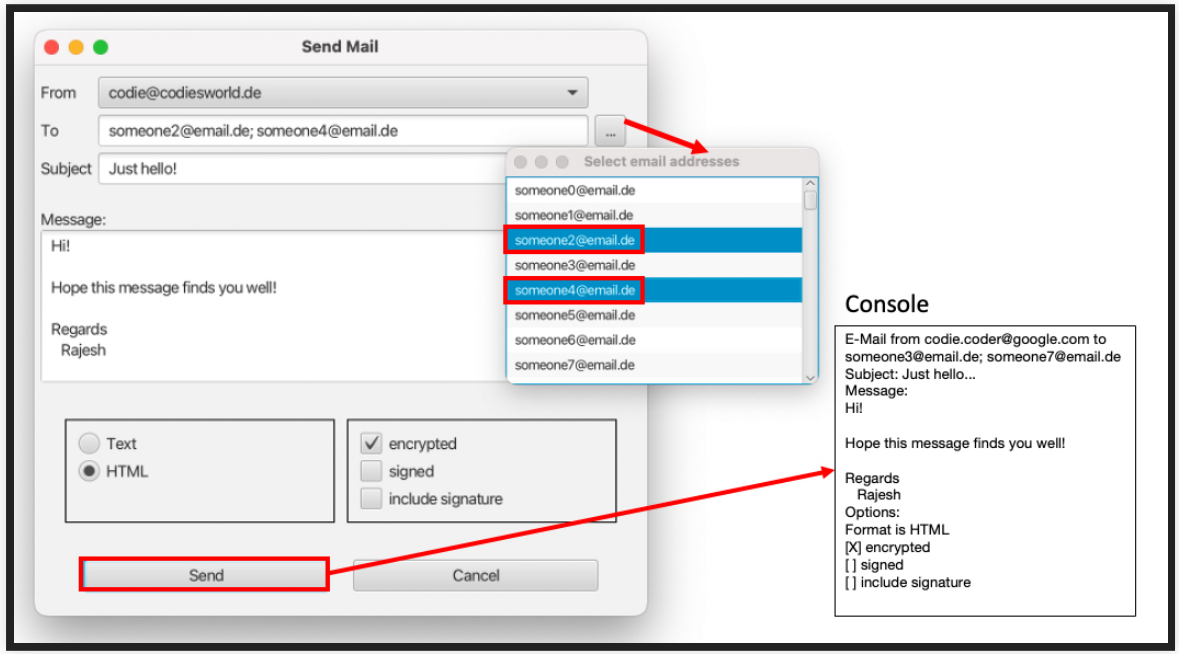
In this section a JavaFX mail dialog is developed as an example to demonstrate the aspects **Controls, Layout and Styling**.



On the next page you will find some information on downloading and installing the application.

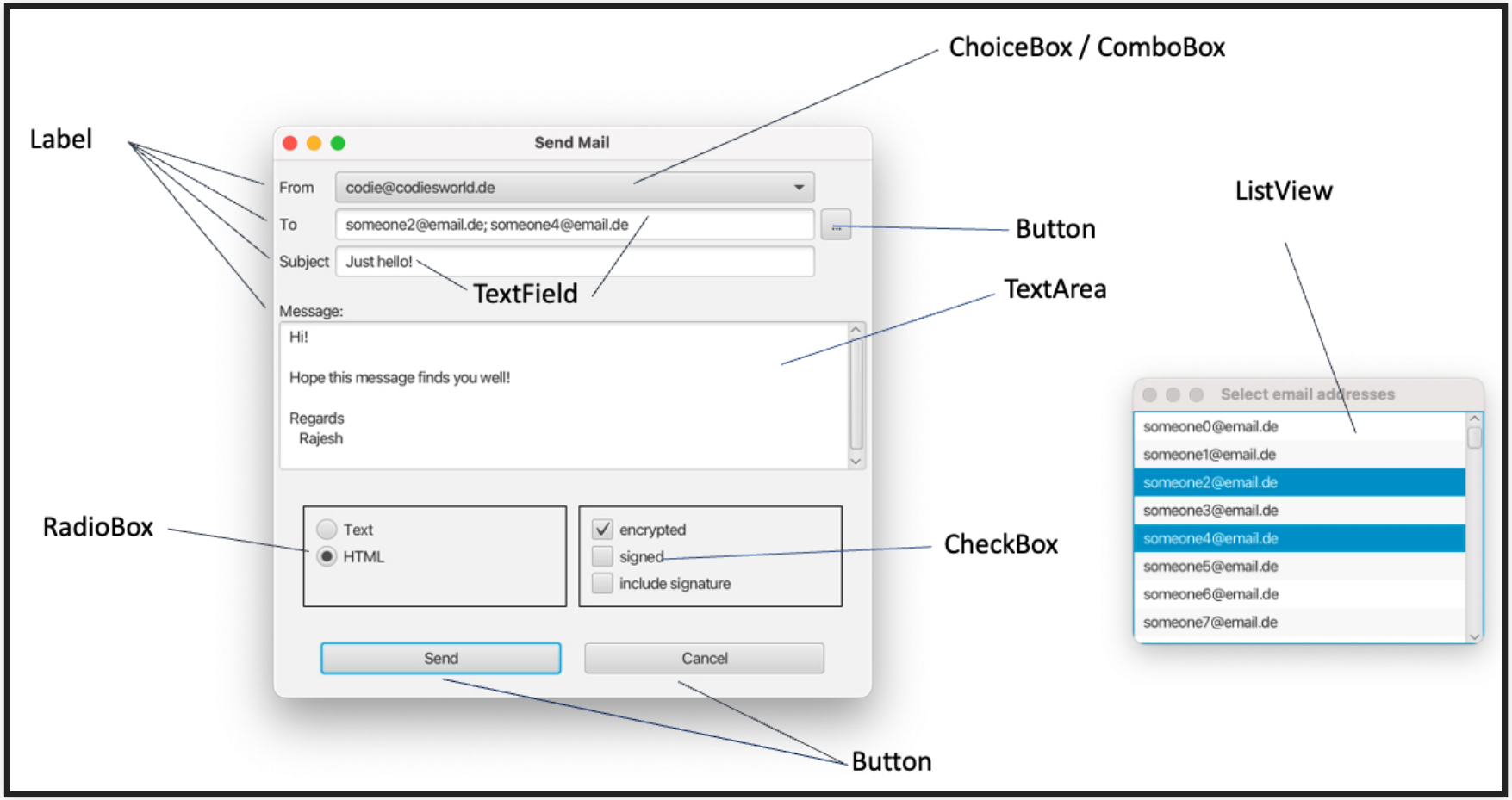
Unzip javafx-mail-gui.zip and import the folder as "Existing Maven Project".

HOW IT WORKS...



- select the sender's email address
- enter or select one or more receiver's email addresses
- enter your message
- choose options
- send email: instead of really sending an email we print out all inputs

WHICH CONTROLS ARE USED?



TOP LEVEL LAYOUT: BORDERPANE

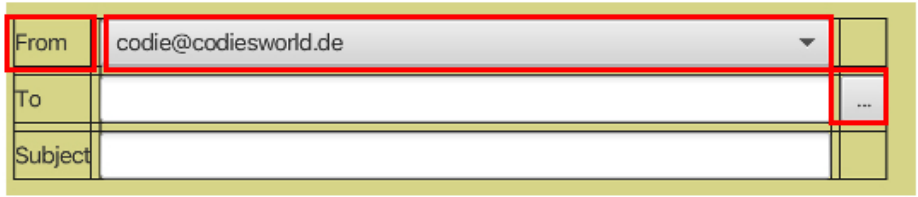
Advantage of using a BorderPane: the center area resizes when the window is being resized!

Java:

```
BorderPane mainPane = new BorderPane();
mainPane.setTop(initTopPane(stage1));
mainPane.setCenter(initMessagePane());
mainPane.setBottom(initBottomPane());
```

LAYOUT OF TOP PANE

GridPane
at position top



3 rows

3 columns

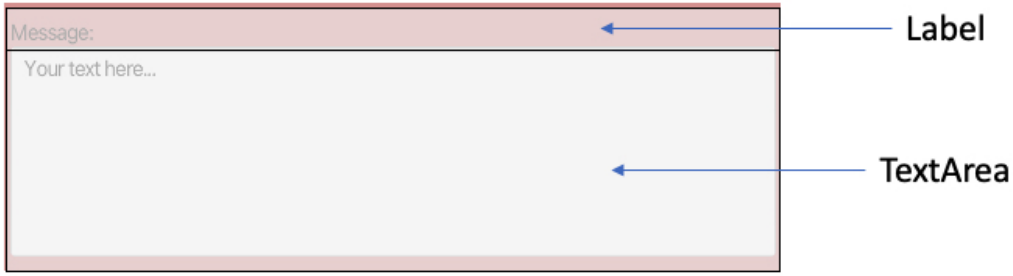
Column Row

Java:

```
grid.add(new Label("From"),        0, 0);
grid.add(new Label("To"),          0, 1);
grid.add(new Label("Subject"),     0, 2);
grid.add(fromChoiceBox,            1, 0);
grid.add(toTextField,              1, 1);
grid.add(subjectTextField,         1, 2);
grid.add(anAuswahlButton,          2, 1);
```

LAYOUT OF CENTER PANE

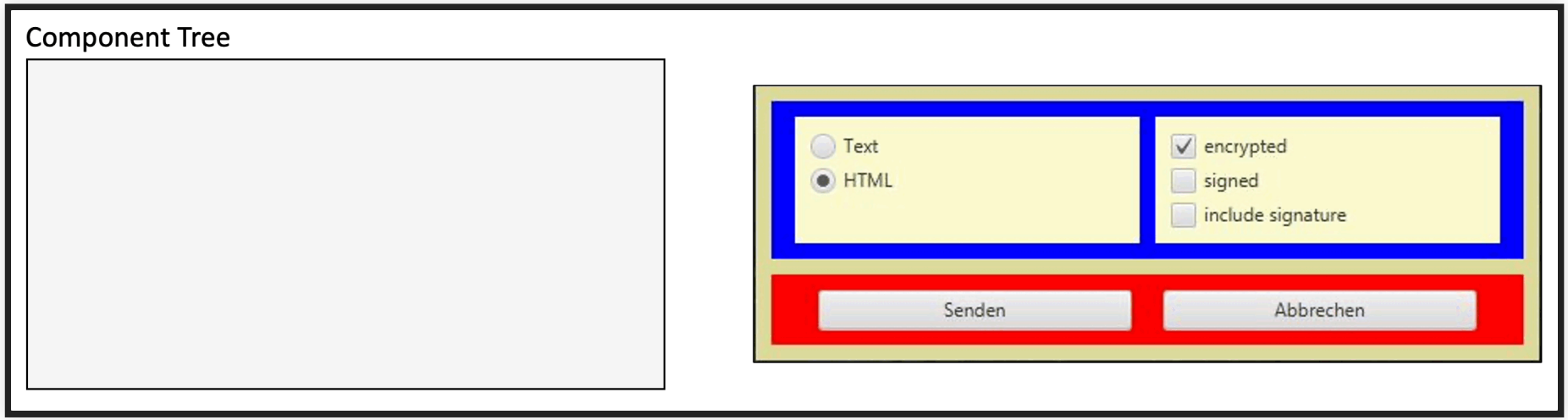
VBox at
center position



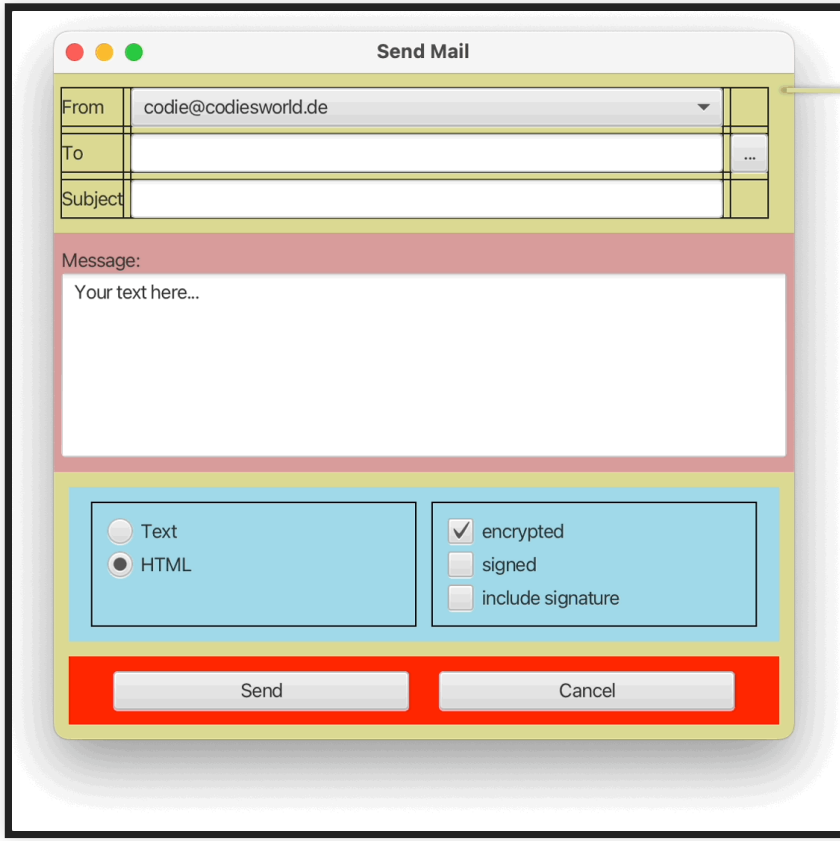
Java:

```
messageTextArea = new TextArea("Your text here...");
pane.getChildren().addAll(new Label("Message:"), messageTextArea);
```

LAYOUT OF BOTTOM PANE

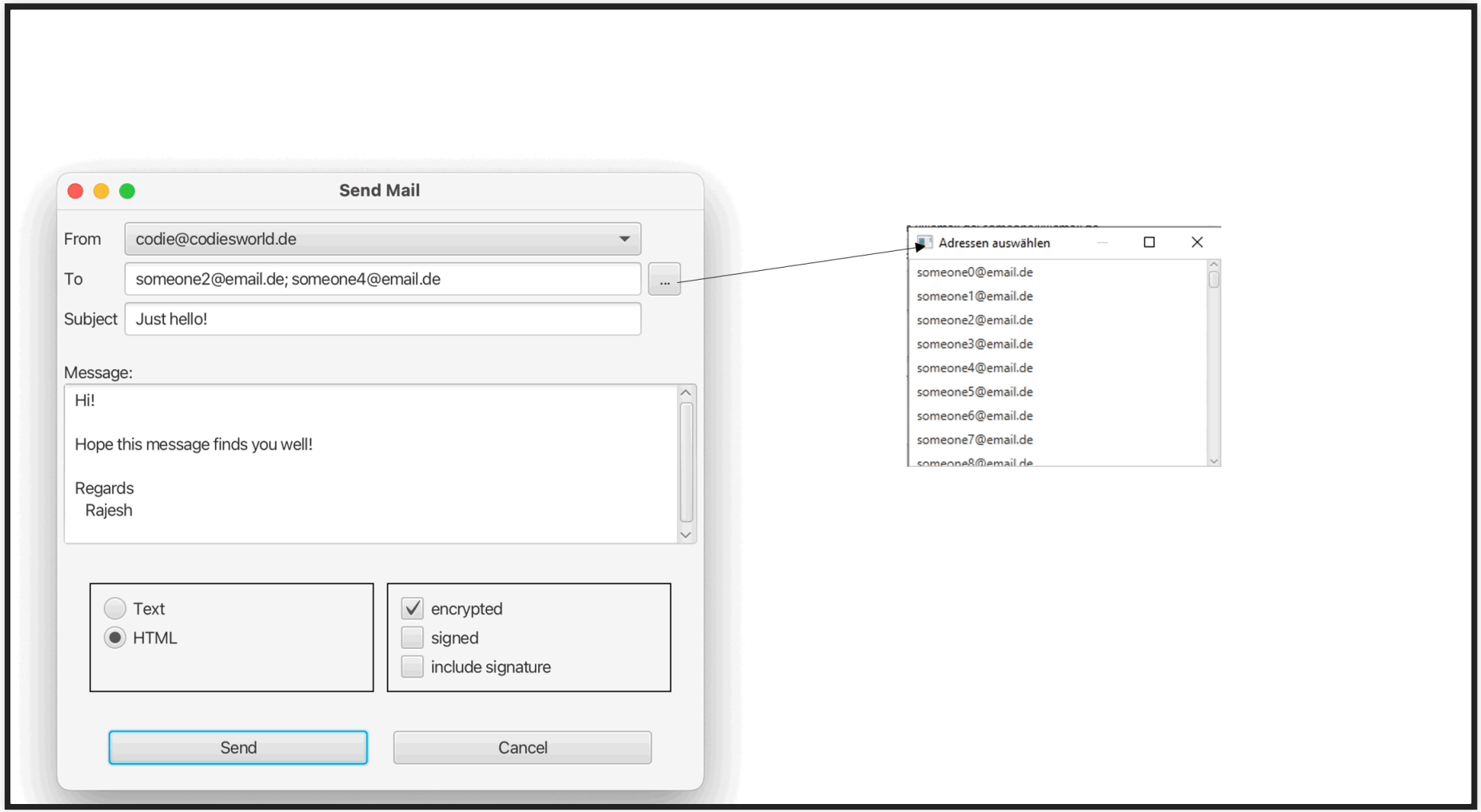


STYLING



- Background color for pane
  here: using CSS style attribute for background-color
  grid.setStyle("-fx-background-color: #dbda9a");

EVENT HANDLING



9.7 EXERCISE FOR SECTIONS 9.1 TO 9.3

WINDOWS AND EVENT HANDLING

Unzip label-clicker.zip and import the folder as "Existing Maven Project".

TASK DEFINITION

In this exercise, you programm a simple window with a label.

When you click on the label, the label text should change from "Servus!" to "Howdy!". The next click, the text should change back to "Servus!" and so on.

Each click should also cause a console output:

```
Label Servus! clicked at position 8/7
Label Howdy! clicked at position 9/7
Label Servus! clicked at position 19/7
```

STEP 1 - BUILD UI

First develop the user interface in the following steps:

1. create a class *LabelClicker* that inherits from *Application* and should get an attribute named *label* for our label. Also define a getter for *label* for the tests.
2. create a class `Main` with a *main* method, call *Application.launch(LabelClicker.class)* in this class.
3. calling *launch* leads to calling the *start* method in which the label is created and displayed. Use the following two lines for this. *pane* is a FlowPane that is placed in the scene and to which the label is added beforehand:

```
pane.setAlignment(Pos.CENTER); // center label in FlowPane
Scene scene = new Scene(pane, 300, 200); // Window size 300 x 200 pixels
```

STEP 2 - ADD EVENT HANDLING

Now program the handling of the click event for the label:

- To do this, pass a lambda to the label instance using the *setOnMouseClicked* method, which changes the label text and performs the console output.
- The label text can be set or changed using the *setText(new label text)* method on the label instance.
- The label text can be read with *getText* to the label instance. This can be used to implement the change logic.
- The event parameter can be used for the output (see task definition). The position of the mouse click can be queried with the *getX* and *getY* getters. Convert position values to integers.

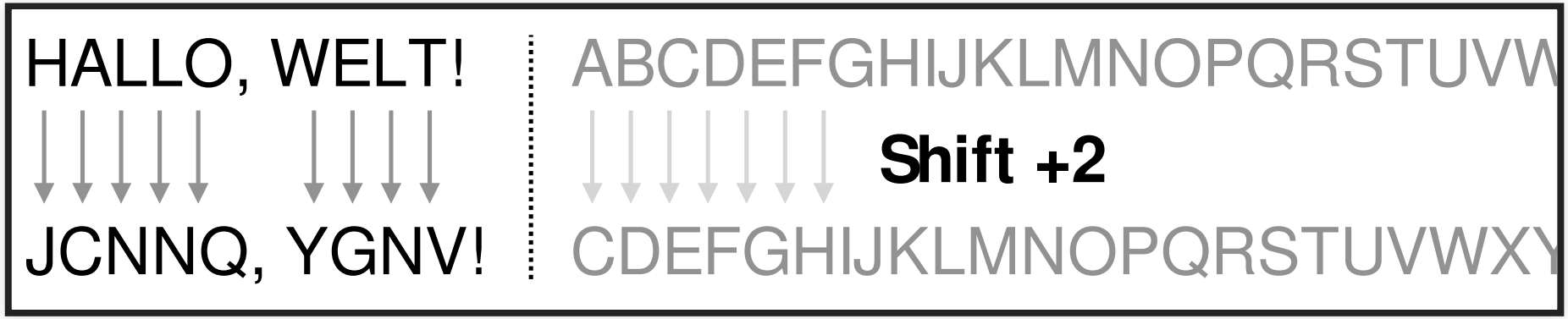
EXECUTING AND TESTING THE APPLICATION

1. start the application via the 'Main' class in the project and select "Run As -> Java Application".
2. start the JUNIT tests in the *test* folder.

9.8 EXERCISE FOR CHAPTER 9.4 AND 9.5

CAESAR CIPHER

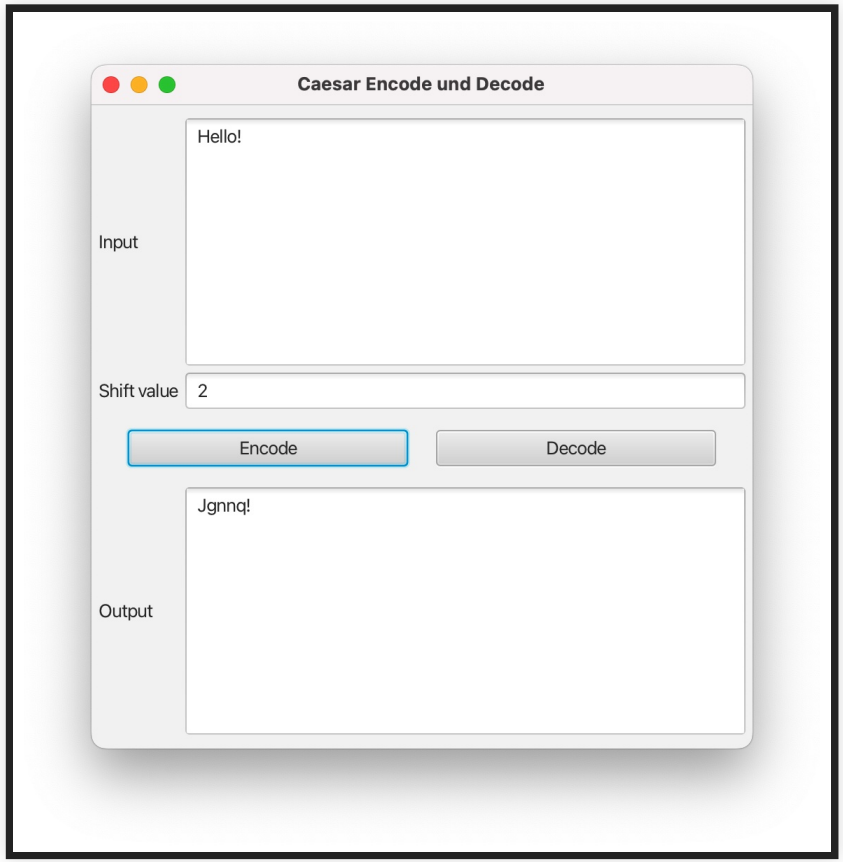
The aim of the exercise is to create a GUI for Caesar encryption with JavaFX.



The Caesar cipher is a simple symmetric encryption method in which the letters of a message are shifted by a certain value. With a shift of 2, `A` becomes `C`.

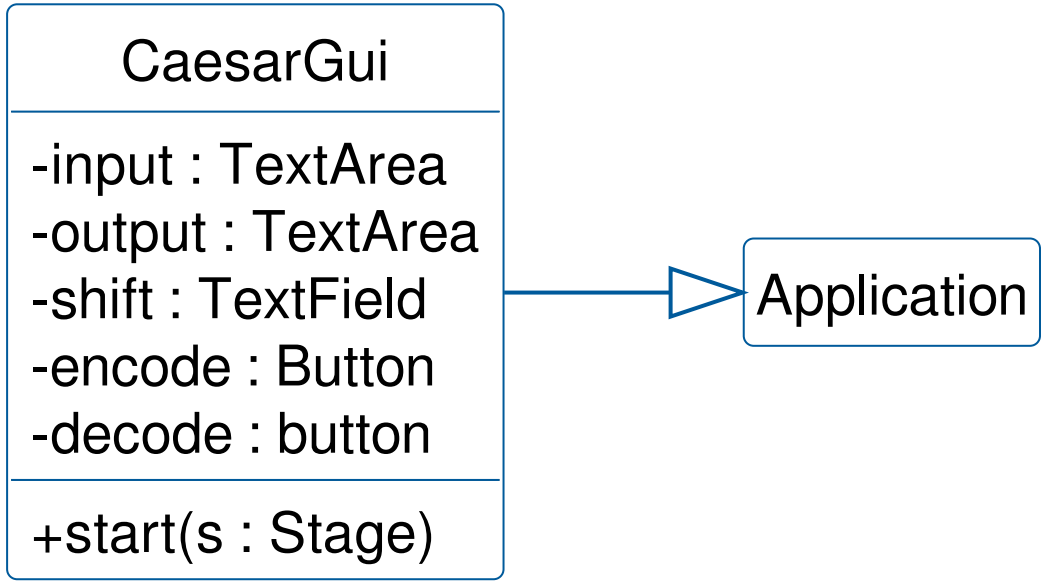
Unzip caesar-gui.zip and import the folder as "Existing Maven Project".

EXERCISE: SURFACE



In the next step, create a GUI that is used for encryption and decryption. You might want to use the control types "TextArea", "TextField" and "Button" for this. Arrange the controls in a suitable layout (see screenshot).

UML class model for the GUI



Please note when implementing

- Create the controls in the `start` method and assign them to the corresponding attributes
- It may be necessary to combine multiple panes.

EXERCISE: APPLY ENCRYPTION

Finally, with regards to event processing for the `encode` and `decode` buttons, use the appropriate implementation of the encryption in the `CaesarEncryption` class.

Start the application via the 'Main' class in the project and select "Run As -> Java Application". Check the solution using the tests.

EXERCISE: APPLICATION OF ENCRYPTION

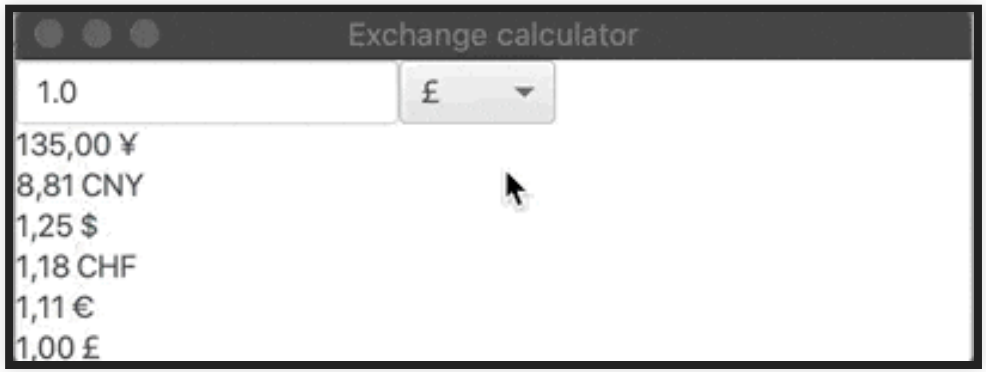
Evaluate the following ciphers, use the one for your application.

- [ ] "Hello, World!" shifted by 10 gives "Rkvvy, Govd!"
- [ ] "Caesar" shifted by 28 results in "Ecguct"
- [ ] "Basics of programming 2" shift by 3 yields "Edvlfv ri surjudpplqj 2"

9.9 EXERCISE CURRENCY CONVERTER

CURRENCY CONVERSION

The following exercise realizes a simple currency converter. The GUI contains an input field and a selection box with the source currency and each update of the input field or the selection box should refresh the values for a set of given target currencies.



Unzip exchange-gui.zip and import the folder as "Existing Maven Project".

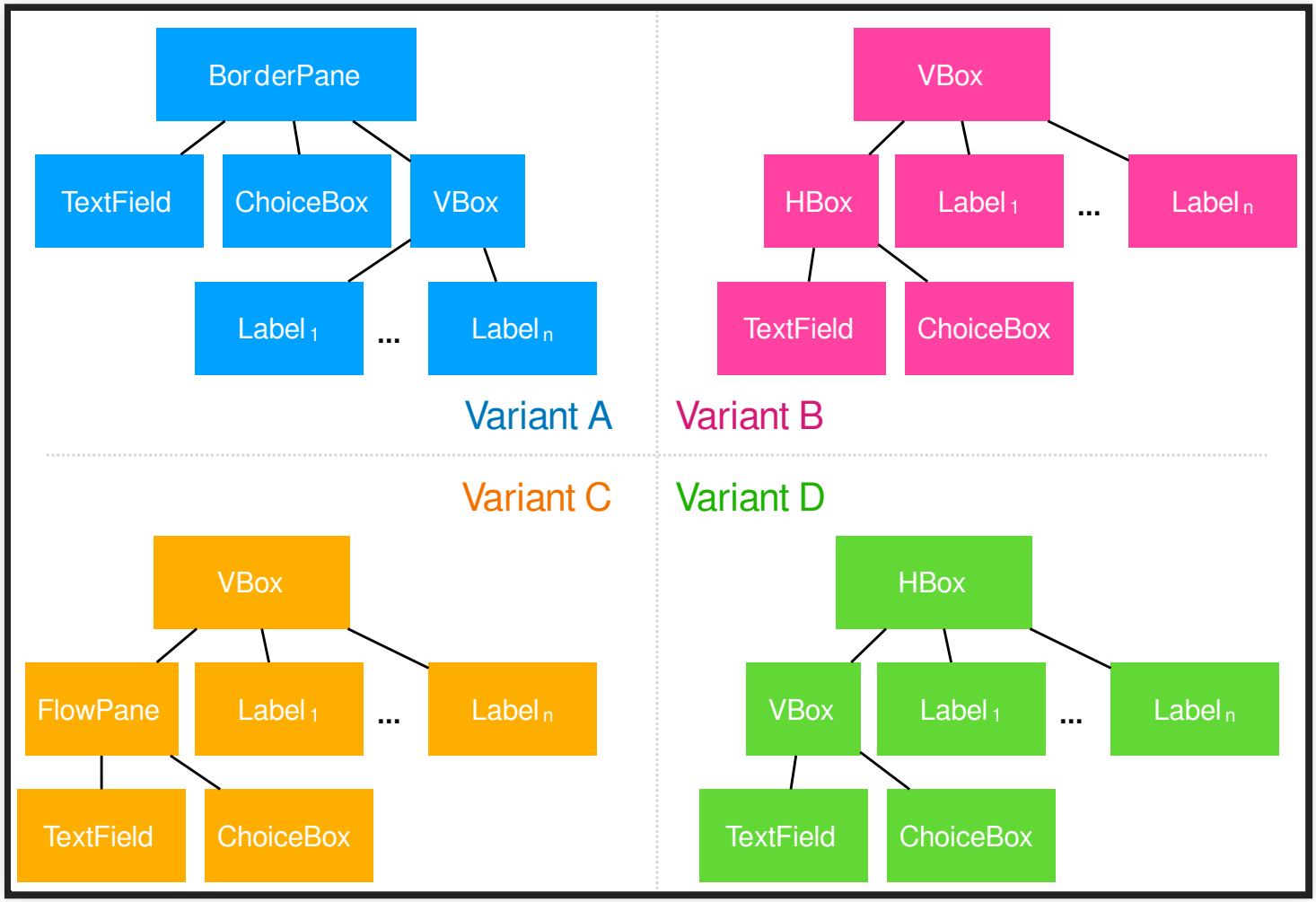
EXERCISE: GUI ELEMENTS

Which controls were used?

- ☐ Label
- ☐ Button
- ☐ TextField
- ☐ ComboBox
- ☐ ChoiceBox
- ☐ TextArea

EXERCISE: FINDING A SUITABLE LAYOUT HIERARCHY

What could a layout hierarchy look like?

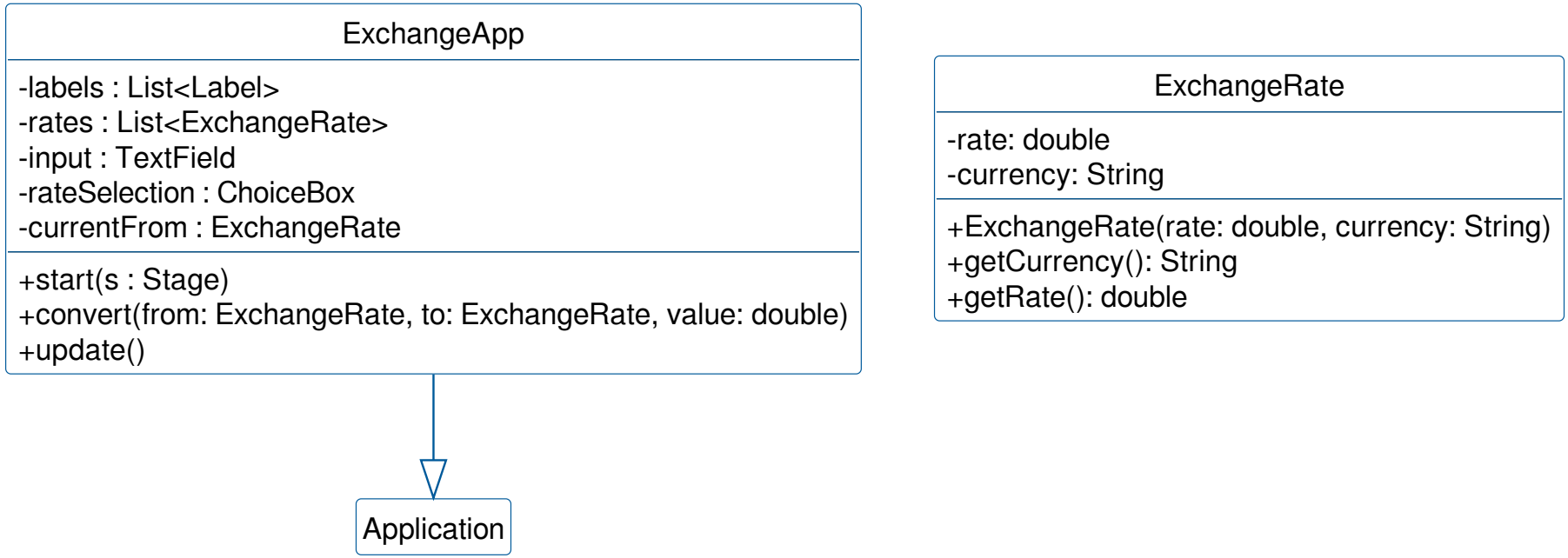


Which variants would be possible?

- [ ] Variant B
- [ ] Variant A
- [ ] Variant C
- [ ] Variant D

EXERCISE: IMPLEMENT EXCHANGE GUI

Look at the classes ExchangeApp and ExchangeRate below.



Realize the GUI controls

- Create the input field and assign it to the `input` attribute
- Create the selection box (ChoiceBox with concrete type String) and assign it to the attribute `rateSelection`
- Iterate over all entries in the `rates` list and create the same number of label instances, add them to the `labels` list

Realize the GUI layouts

- Add input field and selection box into a FlowPane
- Add FlowPane and Labels of `rates` list into a VBox

Finalizing the surface

- Create a scene using VBox
- Put a name on the stage
- Set the scene in the stage
- Show the stage

CONVERSION

Realize the convert method

- Implement conversion from one currency to another using the parameters (all conversion factors use Euro as a reference currency)
- Example:
  - `2.0 USD in EUR` would be `1.7706949978 EUR`
  - If `1 EUR is equal to 1.1295 USD`
  - Calculated according to `2.0 USD / 1.1295 USD * 1.0 EUR`

UPDATING THE LABELS

Realize the update method: update all label fields in the labels list

- Obtain the current initial value from the input field
- Convert the value to a double: Double.parseDouble can be used here
- In a loop over the number of output labels or conversion rates:
  - Call the `convert` method
  - Put the result in the corresponding label

FILL CHOICEBOX

Realize the choices of ChoiceBox

- Use the `setItems` on the selection box - you need an instance of type ObservableList
  - Please be aware 1: The `currency` of the `ExchangeRate` should be used as a string
  - Please be aware 2: You can use the FXCollections class and the static observableList method to convert a known ArrayList into an ObservableList
  - Question: how can you convert the ArrayList into an ObservableList?
- Set the selection to the first element

CHANGE SELECTION

Realize EventHandling for the selection box

- The logic when changing the selection should:
  - Obtain the current selection
  - Obtain the appropriate instance of the ExchangeRate class from the ArrayList `rates`
  - Store the instance in `currentFrom`
  - Call the `update` method

PROCESS INPUT CHANGES

Realize the EventHandling for the input field

- The method `setOnAction` would not be appropriate here
- Better is `setOnKeyReleased`, which is triggered when any key is released
- Call the `update` method

INITIALIZING THE GUI

Configure the GUI so that a value is initially entered in the input field, an element of the selection box is selected and the GUI is updated using the `update` method.

SORTING (OPTIONAL)

Sort the exchange rates using a comparator