Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Module BTI7055: Object-Oriented Programming 2

J.-P. Dubois & S. Kramer
BFH-TI: Computer Science Division / 2017-2018

Graphical User Interfaces

jean-paul.dubois@bfh.ch

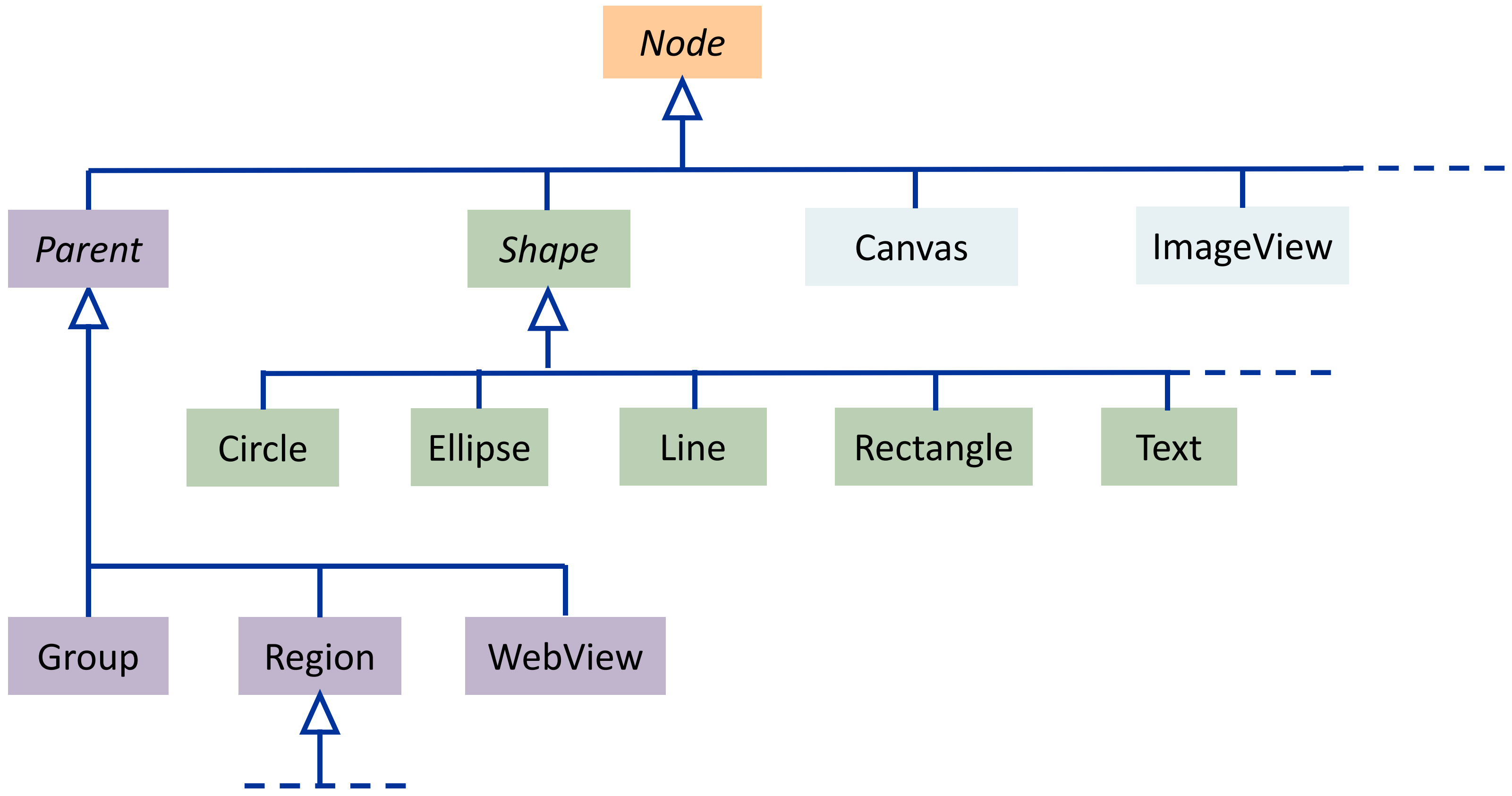
Outline

- Layout Management
- UI Controls
- Events
- Properties and Bindings
  - The JavaFX Properties
  - Bindings
  - Listeners
- Model-View-Controller (MVC)

Outline

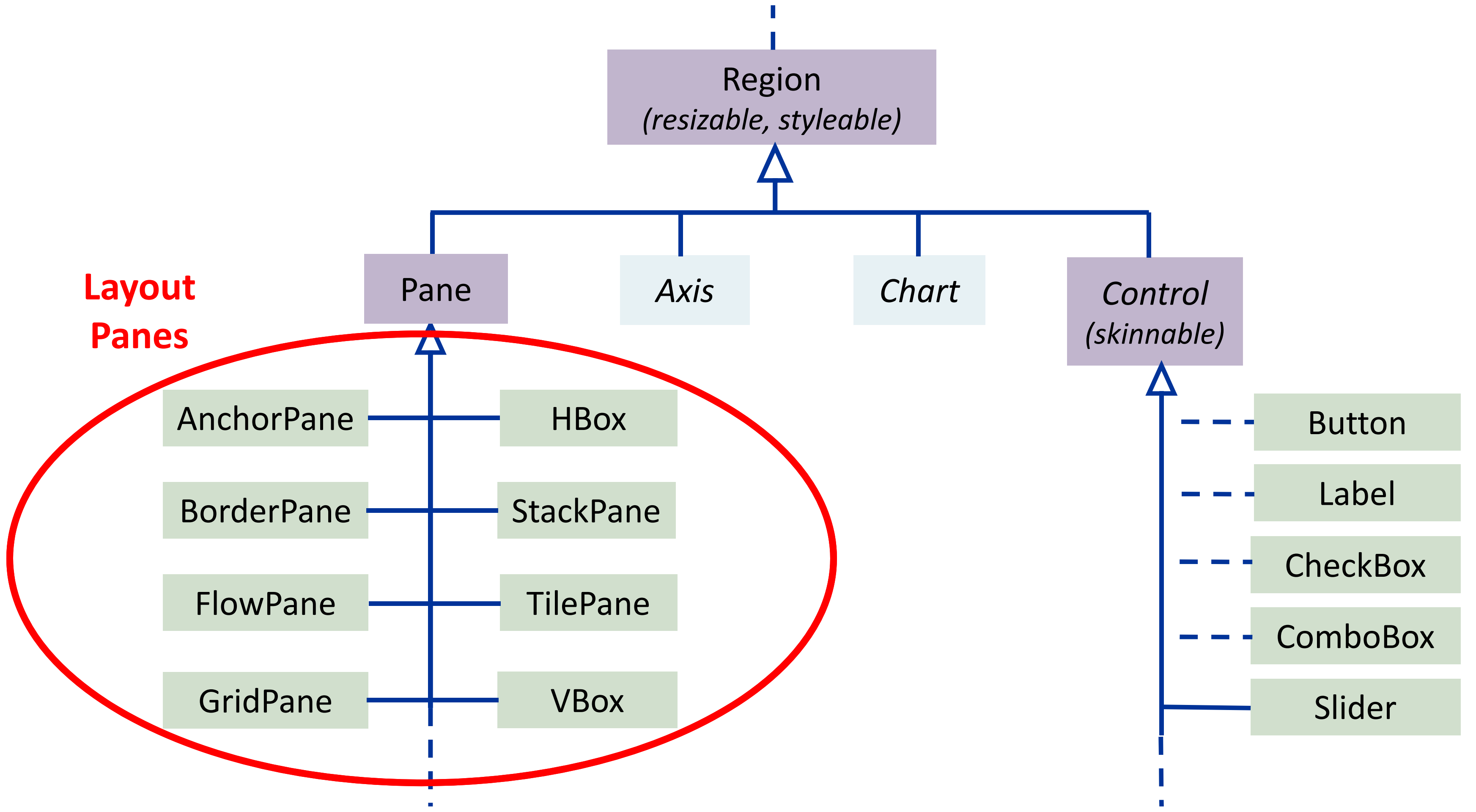
Layout Management

- JavaFX Node Hierarchy (excerpt)



Layout Management

- JavaFX Node Hierarchy 2 (excerpt)



Layout Management

- User-interface components are arranged by placing them inside layout panes

- JavaFX provides several built-in layout panes for the easy setup and management of its components.
  - Layout panes can be nested, allowing you to combine freely various layouts in one window.

- As a window is resized, the layout pane automatically repositions and resizes the nodes that it contains according to the properties for the nodes.

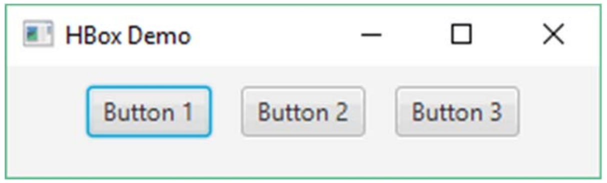
- The JavaFX built-in layout panes (in `javafx.scene.layout`) are:

```
HBox        VBox        BorderPane     AnchorPane
FlowPane    StackPane   TilePane       GridPane
```

Layout Management

- HBox
  - Allows to arrange a series of nodes in a single row.
    - Padding: distance between the nodes and the edges of the pane
    - Spacing: distance between the nodes

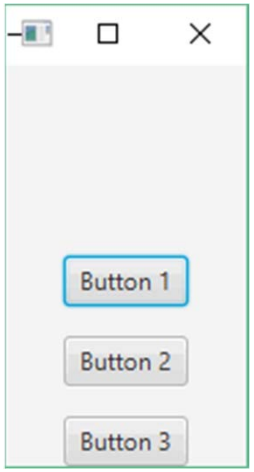
```
Button b1 = new Button("Button 1");
Button b2 = new Button("Button 2");
Button b3 = new Button("Button 3");
HBox pane = new HBox(b1, b2, b3);
pane.setPadding(new Insets(10,40,20,40)); // Top, right, bottom, left
pane.setSpacing(15);
Scene scene = new Scene(pane);
```



Layout Management

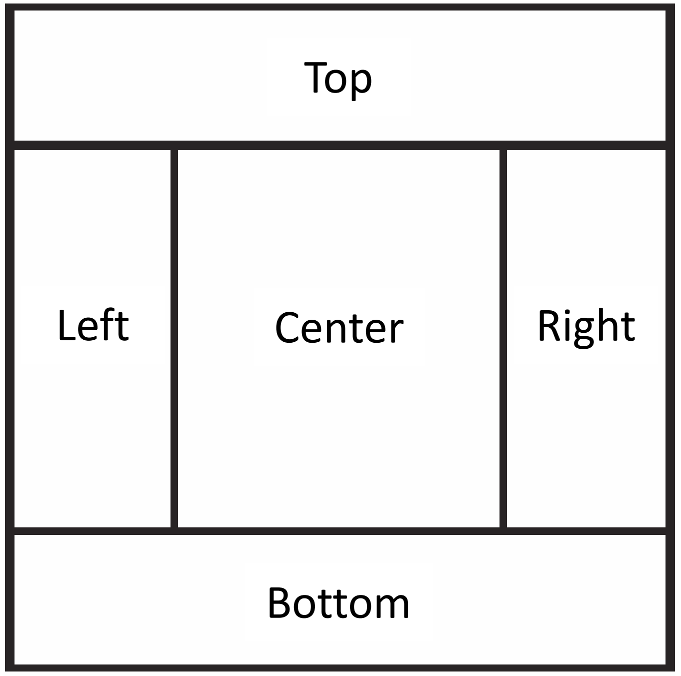
- VBox
  - similar to HBox, but the nodes are arranged in a single column.
    - Alignment: position of the nodes inside the pane

```
Button b1 = new Button("Button 1");
Button b2 = new Button("Button 2");
Button b3 = new Button("Button 3");
VBox pane = new VBox(b1, b2, b3);
pane.setSpacing(15);
pane.setAlignment(Pos.BOTTOM_CENTER);
Scene scene = new Scene(pane, 100, 200);
```



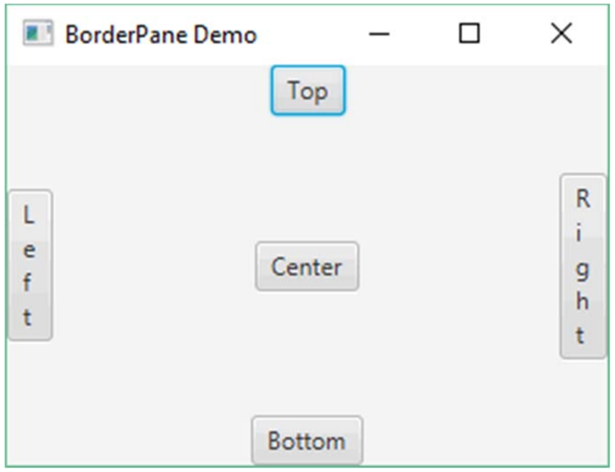
Layout Management

- BorderPane
  - Provides 5 regions in which to place nodes: top, bottom, left, right, and center.
    - The center region can expand horizontally and vertically
    - The border regions (top, right, bottom, left) expand only horizontally, resp. vertically
    - The regions can be any size. If you don’t need one of the regions, you don't need to define it.
  - Often used as basic layout for the main window



Layout Management

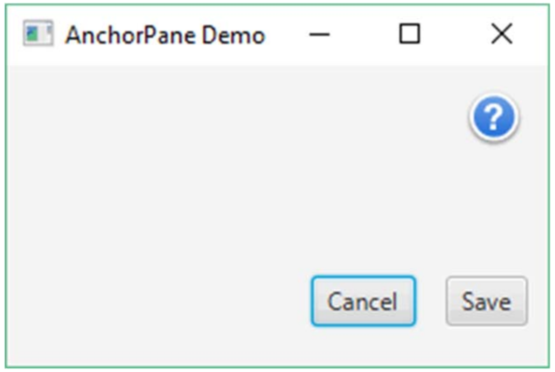
```
Button top = new Button("Top");
BorderPane.setAlignment(top, Pos.CENTER);
Button right = new Button("R\ni\ng\nh\nt");
BorderPane.setAlignment(right, Pos.CENTER);
Button bottom = new Button("Bottom");
BorderPane.setAlignment(bottom, Pos.CENTER);
Button left = new Button("L\ne\nf\nt");
BorderPane.setAlignment(left, Pos.CENTER);
Button center = new Button("Center");

BorderPane pane = new BorderPane();
pane.setTop(top);
pane.setRight(right);
pane.setBottom(bottom);
pane.setLeft(left);
pane.setCenter(center);

Scene scene = new Scene(pane, 300, 200);
```



Layout Management

- AnchorPane
  - To anchor nodes to the top, bottom, left side or right side of the pane.
  - As the window is resized, the nodes maintain their position relative to their anchor point.

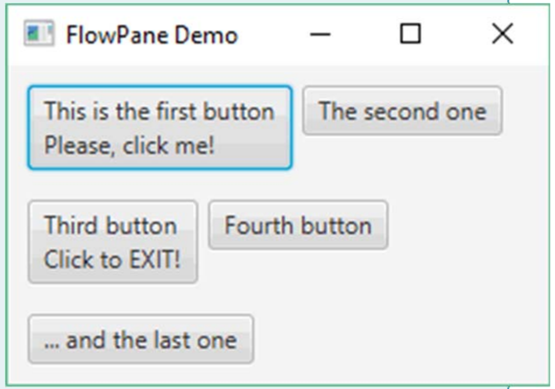
```
Button save = new Button("Save");
Button cancel = new Button("Cancel");
HBox buttons = new HBox(15, cancel, save);
AnchorPane.setRightAnchor(buttons, 10.0);
AnchorPane.setBottomAnchor(buttons, 20.0);

ImageView img = new ImageView(getClass()
        .getResource("qmark.png").toExternalForm());
AnchorPane.setRightAnchor(img, 10.0);
AnchorPane.setTopAnchor(img, 10.0);

AnchorPane pane = new AnchorPane(buttons, img);
Scene scene = new Scene(pane, 270, 150);
```



Layout Management

- FlowPane
  - Lays out all nodes consecutively and wraps at the pane boundary
  - Nodes can flow horizontally (in rows) or vertically (in columns)
    - Vgap / Hgap: Spacing between rows (vert.) and columns (hor.)
    - prefWrapLength: Preferred with/height where content should wrap

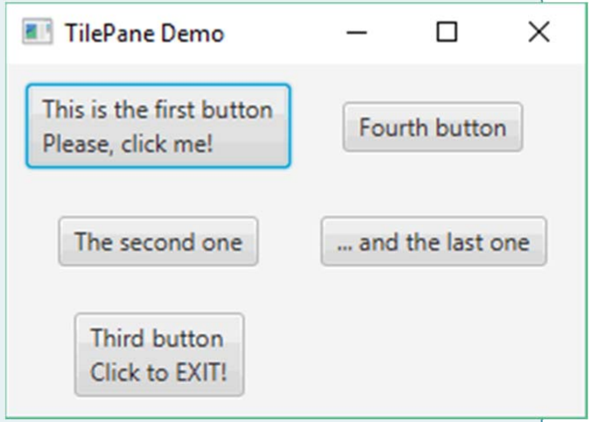
```
Button b1 = new Button("This is the first button\nPlease, click me!");
Button b2 = new Button("The second done");
Button b3 = new Button("Third Button\nClick to EXIT!");
Button b4 = new Button("Fourth button");
Button b5 = new Button("... and the last one");
FlowPane pane = new FlowPane(b1, b2, b3, b4, b5);
pane.setPadding(new Insets(10));
pane.setVgap(15);
pane.setHgap(5);
pane.setPrefWrapLength(280);
Scene scene = new Scene(pane);
```



Layout Management

- TilePane
  - Like FlowPane, but each cell (tile) has the same size
  - The cells have the size to put the biggest object in
    - PrefRows / PrefColumns: preferred number of rows / columns for a vertical / horizontal TilePane

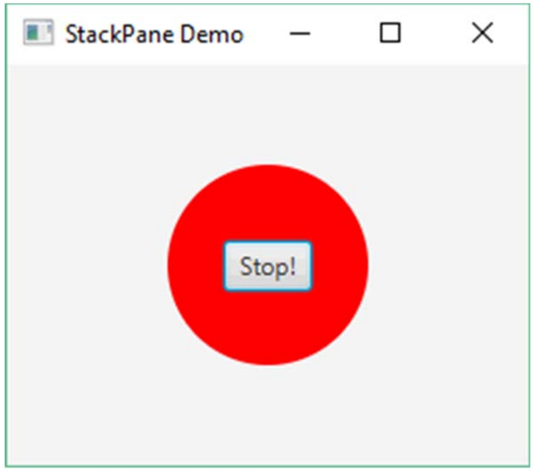
```
Button b1 = new Button("This is the first button\nPlease, click me!");
Button b2 = new Button("The second done");
Button b3 = new Button("Third Button\nClick to EXIT!");
Button b4 = new Button("Fourth button");
Button b5 = new Button("... and the last one");
TilePane pane = new TilePane(
      Orientation.VERTICAL, b1, b2, b3, b4, b5);
pane.setPadding(new Insets(10));
pane.setVgap(15);
pane.setHgap(5);
pane.setPrefRows(3);
Scene scene = new Scene(pane);
```



Layout Management

- `StackPane`
  - Places all of the nodes within a single stack with each new node added on top of the previous node.
  - Per default, in a `StackPane` the nodes are centered

```
Button button = new Button("Stop!");
Circle circle = new Circle(50);
circle.setFill(Color.RED);
StackPane pane = new StackPane(circle, button);
Scene scene = new Scene(pane, 260, 200);
```



Layout Management

- `GridPane`
  - Allows to built flexible layouts that are organized in rows and columns.
  - Consists on a flexible grid of rows and columns
  - The nodes can be placed in any cell in the grid
  - The nodes can span cells
  - The cells adapt dynamically to the size of the nodes
  - The cells may be empty
  - To help debugging, the grid lines can be made visible

Layout Management

- Adding a node to a `GridPane`:
  - To add a node in the grid, specify the column (first) and the row (last) indices (counting from 0) :

```
pane.add(image, 3, 0); // Place image at position (0;3)
```

  - If the node spans cells, also specify the number of columns (first) and the number of rows (last) it should span

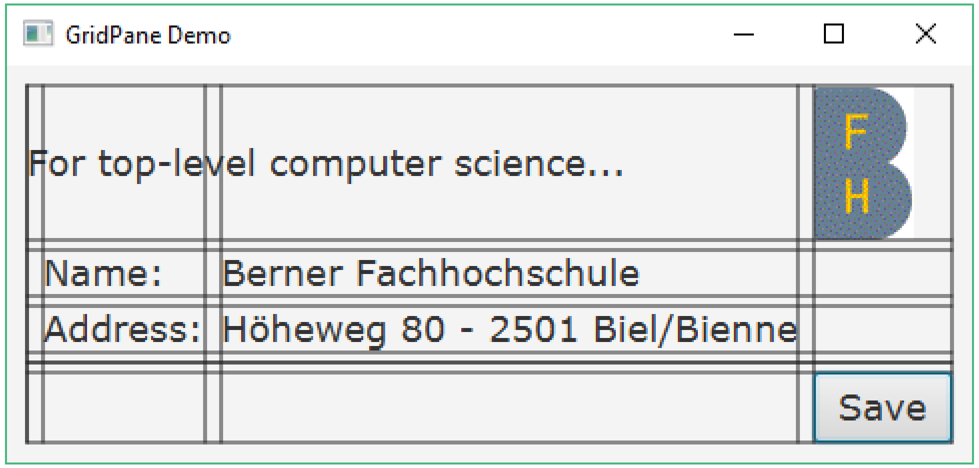
```
// Place a label that spans 1 row and 3 columns at pos. (0;0)
pane.add(label, 0, 0, 3, 1);
```

  - As for other layout panes, you can specify padding values and gaps between the row and columns

```
pane.setVgap(5); // 5 pixels between rows
pane.setHgap(8); // 8 pixels between columns
pane.setPadding(new Insets(10)); // 10 pixels around the grid
```

Layout Management

- A `GridPane` with visible grid lines
  - Note that the 3rd line is empty



GridPane Demo

For top-level computer science...

Name:	Berner Fachhochschule
Address:	Höheweg 80 - 2501 Biel/Bienne

Layout Management

- A `GridPane` with visible grid lines (code excerpt)

```
GridPane pane = new GridPane();
pane.setVgap(5);
pane.setHgap(8);
pane.setPadding(new Insets(10));
pane.setStyle("-fx-font: 18 Verdana"); // Set pane font

ImageView logo = new ImageView(getClass()
                              .getResource("logo_bfh.gif").toExternalForm());
pane.add(logo, 3, 0);
Label heading = new Label("For top-level computer science...");
pane.add(heading, 0, 0, 3, 1);
pane.add(new Label("Name:"), 1, 1);
pane.add(new Label("Berner Fachhochschule"), 2, 1);
pane.add(new Label("Address:"), 1, 2);
pane.add(new Label("Höheweg 80 - 2501 Biel/Bienne"), 2, 2);
pane.add(new Button("Save"), 3, 4);

Scene scene = new Scene(pane);
pane.setGridLinesVisible(true); // Make the grid lines visible
```

Layout Management

- Row and Column Constraints:
  - You can change the standard behavior of the `GridPane` by specifying row or column constraints
  - Examples:
    - Specify a fixed height for row #0

```
pane.getRowConstraints().add(0, new RowConstraints(50));
```

    - Column #1 should be resized to 25% of the `GridPane`'s available width and the remaining space attributed to columns #2.

```
ColumnConstraints cc = new ColumnConstraints();
cc.setPercentWidth(25);
ColumnConstraints cc2 = new ColumnConstraints();
cc2.setHgrow(Priority.ALWAYS)
pane.getColumnConstraints()
  .addAll(new ColumnConstraints(), cc, cc2);
```

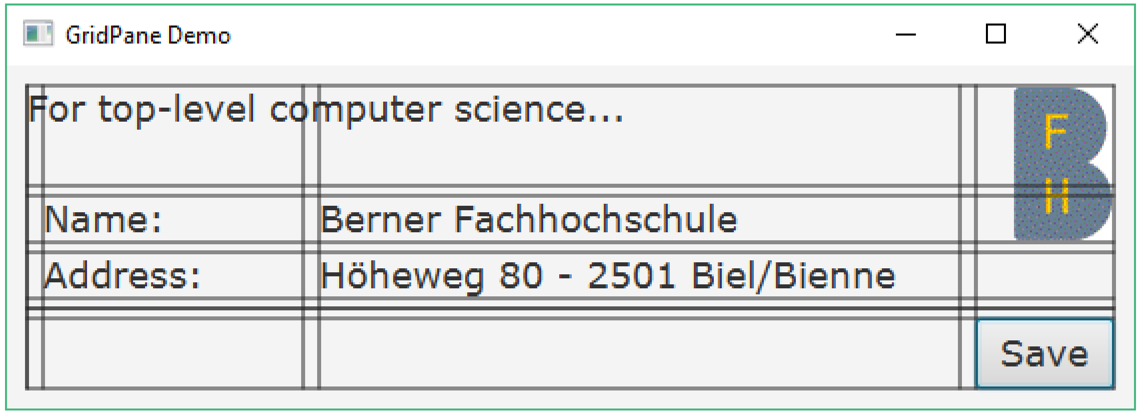
Empty constraint for col. #0

Layout Management

- You can also specify the horizontal and vertical alignment of the nodes in the `GridPane`

```
GridPane.setHalignment(logo, HPos.RIGHT);
GridPane.setValignment(logo, VPos.TOP);
GridPane.setHalignment(heading, HPos.LEFT);
GridPane.setValignment(heading, VPos.TOP);
```

- Layout with the given constraints and alignment (and increased width):


GridPane Demo

For top-level computer science...

Name:	Berner Fachhochschule
Address:	Höheweg 80 - 2501 Biel/Bienne

Layout Management

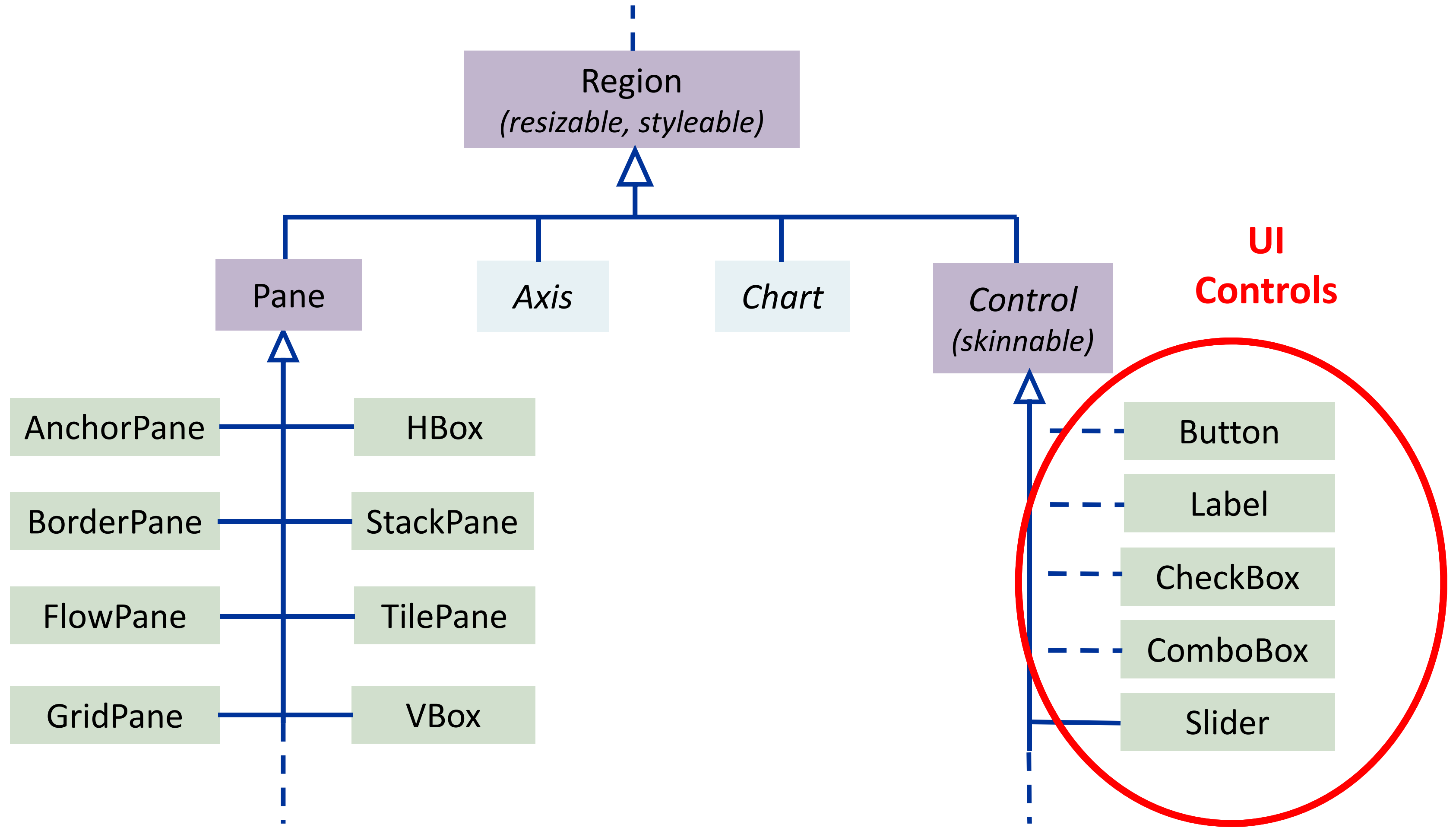
- The calculation of the size of a layout panes and its children according to the selected layout can be rather complicated, It is based on the following values:
  - `minWidth / minHeight`
  - `prefWidth / prefHeight`
  - `maxWidth / maxHeight`

Outline

- Layout Management
- UI Controls
- Events
- Properties and Bindings
  - The JavaFX Properties
  - Bindings
  - Listeners
- Model-View-Controller (MVC)

UI Controls

- JavaFX Node Hierarchy 2 (excerpt)



UI Controls

- Controls are UI components that allow user input or output.

- Some simple controls are presented hereafter, defined (in `javafx.scene.control`) with the classes:
  - Label, Button, CheckBox, RadioButton
  - TextField, PasswordField, TextArea
  - ComboBox
  - Menu

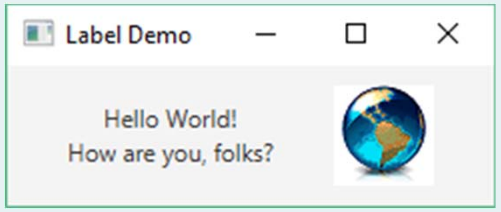
- For deeper explanations and more JavaFX controls, please look at:

```
http://docs.oracle.com/javase/8/javafx/user-interface-tutorial/ui_controls.htm
```

UI Controls

- Labels
  - Labels can contain a text and another node for decoration (mostly an image).

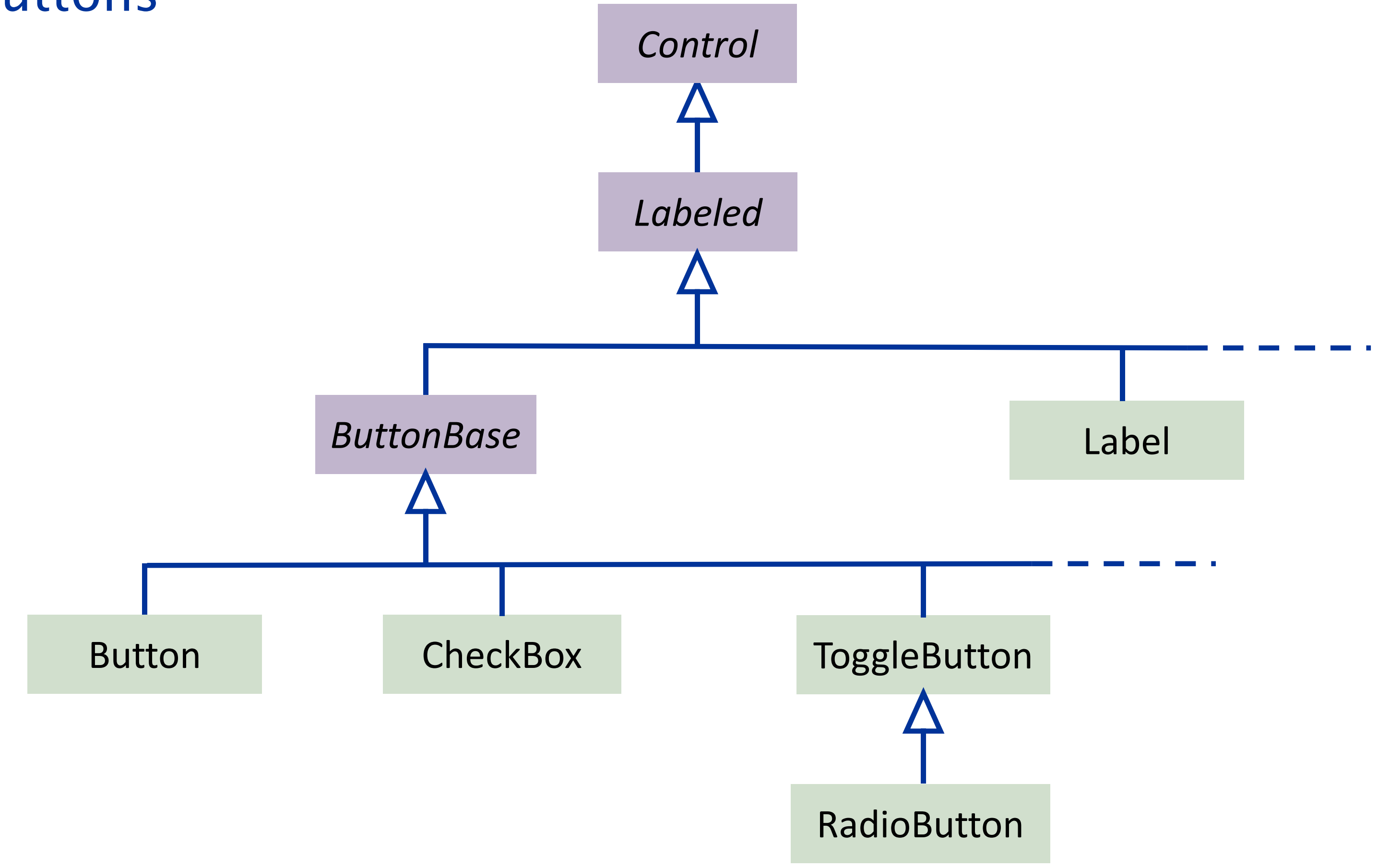
```
Label label = new Label("Hello World!\nHow are you, folks?");
label.setGraphic(new ImageView(getClass()
                 .getResource("world.png").toExternalForm()));
label.setContentDisplay(ContentDisplay.RIGHT);
label.setGraphicTextGap(30);
label.setAlignment(Pos.CENTER);
label.setTextAlignment(TextAlignment.CENTER);
label.setPadding(new Insets(10, 30, 10, 30));
Scene scene = new Scene(label);
```



Hello World!
How are you, folks?

UI Controls

- Buttons

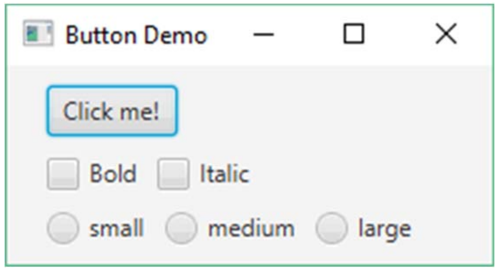


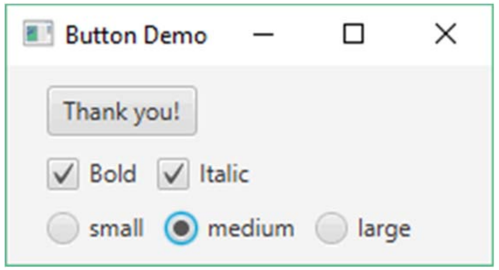
UI Controls

- Buttons
  - Buttons inherit from the same base class Labeled as labels
  - They can also contain a text and another node for decoration.
  - When clicked (or selected), a button generates an ActionEvent
  - JavaFX provides various types of buttons
    - Simple buttons, check boxes, radio buttons, …
  - Check boxes and radio buttons allow to select options
  - Radio buttons can be grouped together.

UI Controls

```
Button simpleButton = new Button("Click me!");
simpleButton.setOnAction(e -> simpleButton.setText("Thank you!"));

CheckBox boldCheckBox = new CheckBox("Bold");
CheckBox italicCheckBox = new CheckBox("Italic");

RadioButton smallButton = new RadioButton("small");
RadioButton mediumButton = new RadioButton("medium");
RadioButton largeButton = new RadioButton("large");
ToggleGroup group = new ToggleGroup();
smallButton.setToggleGroup(group);
mediumButton.setToggleGroup(group);
largeButton.setToggleGroup(group);
VBox root = new VBox(10, simpleButton,
                     new HBox(10, boldCheckBox, italicCheckBox),
                     new HBox(10, smallButton, mediumButton, largeButton));
root.setPadding(new Insets(10, 40, 10, 20));
Scene scene = new Scene(root);
```





UI Controls

- Note that:
  - In a group of radio buttons:
    - Only one button can be selected at a time
    - When a button is selected, the previously selected button in the group is automatically turned off
  - You can programmatically select or unselect a radio button or a check box:

```
      largeButton.setSelected(true); // Select button
```

  - To test whether a radio button or a check box is selected, use:

```
     if (largeButton.isSelected()) ...
```

UI Controls

- Check boxes can provide three states:
  - ☑ Selected
  - ☐ Unselected
  - ⊟ Indeterminate

- To allow the use of the indeterminate state, call:

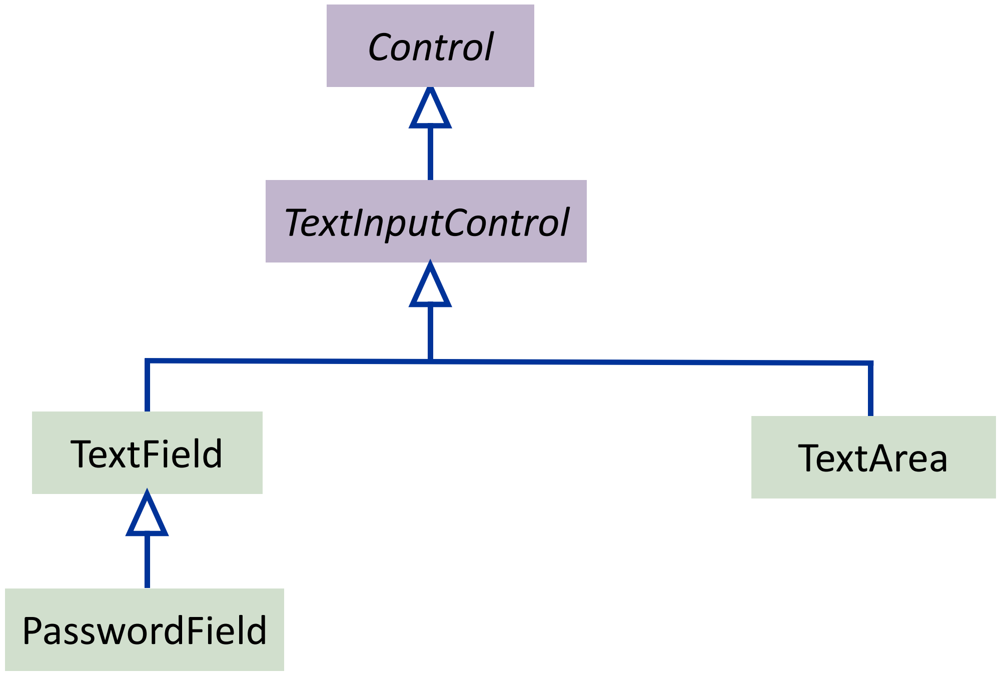
```
    boldCheckBox.setAllowIndeterminate(true);
```

- To work with the indeterminate state, use:

```
  // Select/Unselect indeterminate state
  boldCheckBox.setIndeterminate(...);
  // Test indeterminate state
  if (boldCheckBox.isIndeterminate()) ...
```

UI Controls

- Text input controls



UI Controls

- Text Input Controls
  - To process a *single line* of text, use a `TextField`
  - A `PasswordField` is a text field that allows to enter a password
  - `TextField`'s (and `PasswordField`'s) generate an `ActionEvent` when the Enter key is typed
  - To process a *multiline text*, use a `TextArea`
  - `TextArea`'s support text scrolling

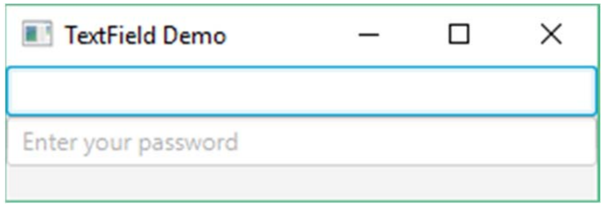
UI Controls

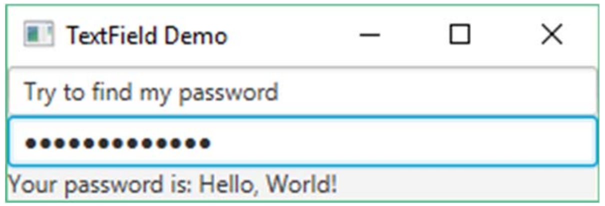
- Some useful methods:
  - For `TextField`'s and `TextArea`'s:
    - `String getText()`: returns the text string contained in the control
    - `setText(String s)`: writes a string into the control
    - `appendText(String s)`: appends a string into the control
    - `replaceText(int from, int to, String s)`: replaces a range of character with the given string
    - `replaceSelection(String s)`: replaces the selection with the given replacement string
    - `setEditable(boolean b)`: determines if the control is editable
    - `setPrefColumnCount(int v)`: defines the preferred control width
  - For `TextArea`'s only:
    - `setPrefRowCount(int v)`: defines the preferred control height
    - `setWrapText(boolean b)`: determines if the text has to be wrapped on a new line when it exceeds the width of the `TextArea`

UI Controls

- Showing `TextFields`:

```
TextField textField = new TextField();
textField.setPrefColumnCount(25);
PasswordField passwordField = new PasswordField();
passwordField.setPromptText("Enter your password");
Label label = new Label();
passwordField.setOnAction(e ->
   label.setText("Your password: " + passwordField.getText()));
VBox pane = new VBox(textField, passwordField, label);
Scene scene = new Scene(pane);
```



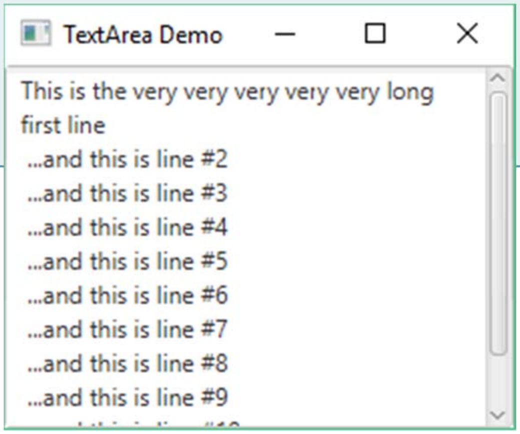


Your password is: Hello, World!

UI Controls

- Showing a `TextArea`:

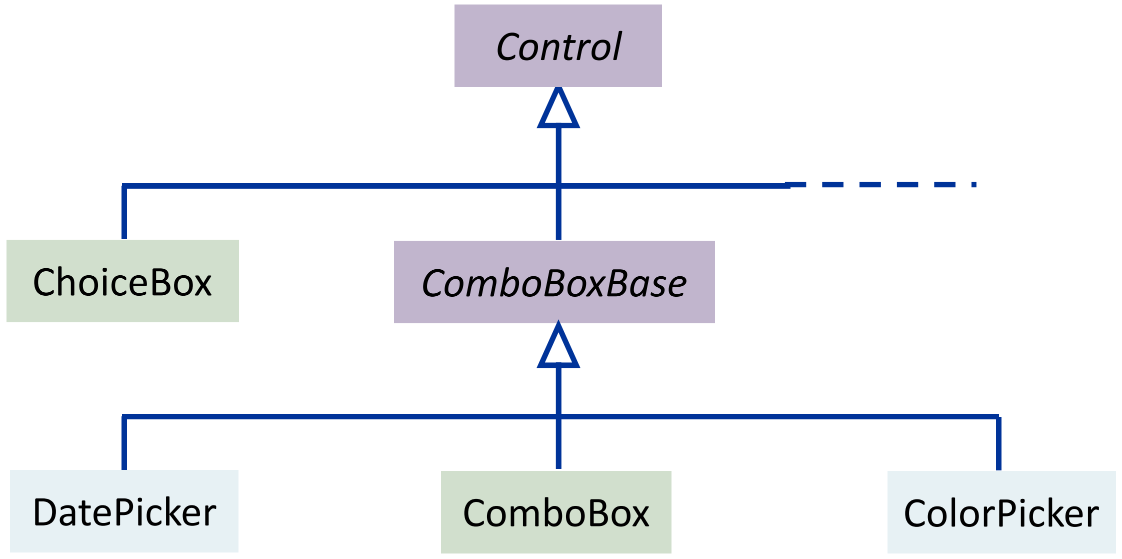
```
TextArea textArea = new TextArea();
textArea.setPrefRowCount(10);
textArea.setPrefColumnCount(20);
textArea.setText("This is the very very very very very"
                 + " long first line");
for (int i = 2; i < 12; i++)
  textArea.appendText("\n ...and this is line #" + i);
textArea.setWrapText(true);
textArea.setEditable(false);
Scene scene = new Scene(textArea);
```



```
This is the very very very very very long
first line
 ...and this is line #2
 ...and this is line #3
 ...and this is line #4
 ...and this is line #5
 ...and this is line #6
 ...and this is line #7
 ...and this is line #8
 ...and this is line #9
```

UI Controls

- Combo Boxes



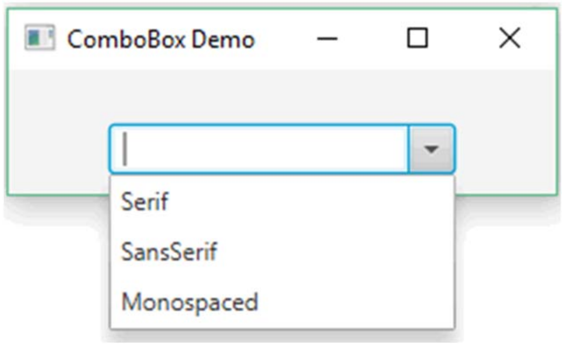
UI Controls

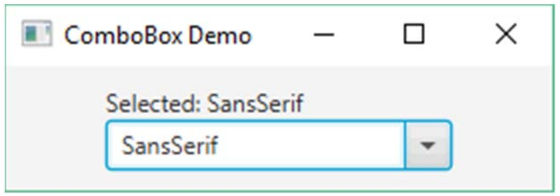
- Combo Boxes:
  - When you have to select one of a large set of choices, use a `ComboBox`
  - A combo box is a combination of a drop-down list and a field
    - The user can select a value from the list, which appears on the user's request.
    - The current selection is displayed in the field
  - `ComboBox` is a generic class. Give as type parameter, the type of the objects that are stored in the combo box
  - When a value is selected, or a value entered in the field (editable combo box), a combo box generates an `ActionEvent`
  - A `ChoiceBox` is a simpler form of a `ComboBox`

UI Controls

- Showing a `ComboBox`:

```
ComboBox<String> comboBox = new ComboBox<>();
comboBox.getItems().addAll("Serif", "SansSerif", "Monospaced");
comboBox.setEditable(true);
Label label = new Label();
VBox root = new VBox(label, comboBox);
root.setPadding(new Insets(10, 50, 10, 50));
comboBox.setOnAction(e ->
            label.setText("Selected: " + comboBox.getValue()));
Scene scene = new Scene(root);
```

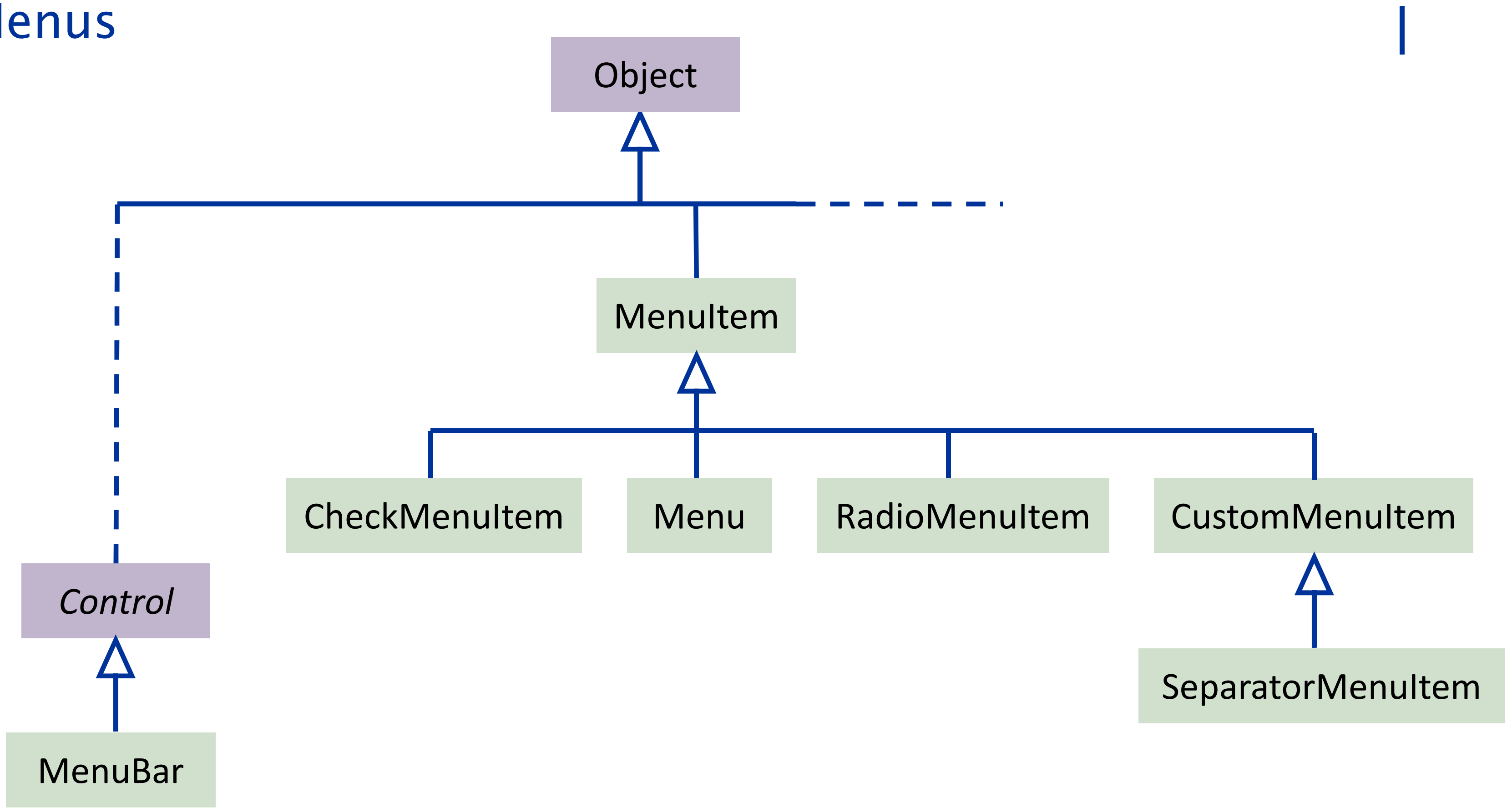




Selected: SansSerif

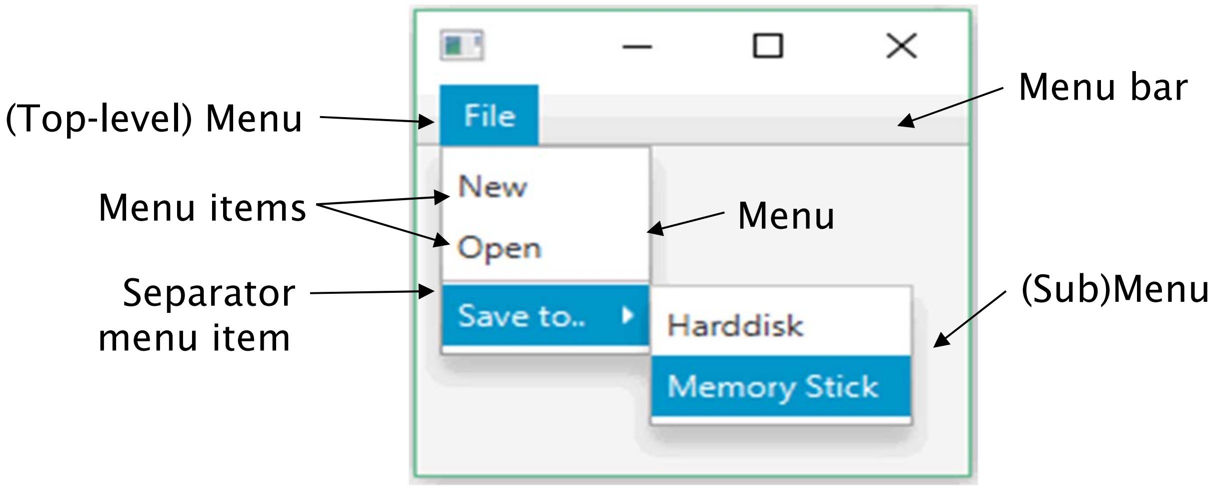
UI Controls

- Menus



UI Controls

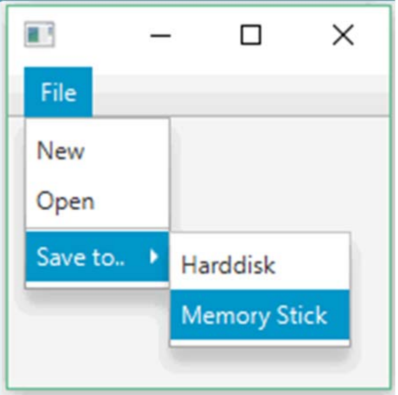
- The menu bar can be placed "anywhere" in a layout pane
- The menu bar contains menus
- A menu contains submenus and menu items
- A menu item has no further submenus



UI Controls

- Building a menu structure

```
MenuBar menuBar = new MenuBar();
Menu fileMenu = new Menu("File");
MenuItem newItem = new MenuItem("New");
MenuItem openItem = new MenuItem("Open");
Menu  saveMenu = new Menu("Save to..");
MenuItem hdItem = new MenuItem("Harddisk");
MenuItem msItem = new MenuItem("Memory Stick");
fileMenu.getItems().addAll(newItem, openItem,
                           new SeparatorMenuItem(), saveMenu);
saveMenu.getItems().addAll(hdItem, msItem);
menuBar.getMenus().add(fileMenu);
BorderPane pane = new BorderPane();
pane.setTop(menuBar);
Scene scene = new Scene(pane, 190, 160);
```



UI Controls

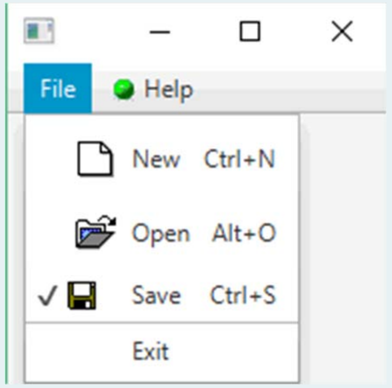
- `CheckMenuItem`'s and `RadioMenuItem`'s:
  - They are menu items that can be toggled between selected and unselected states, in a similar way as the corresponding button types (`CheckMenuItem's` have no indeterminate state).
  - You can insert them in a menu structure like the `MenuItem`'s.

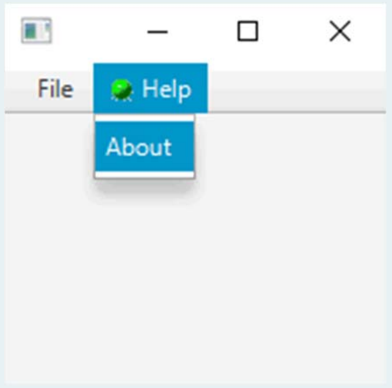
- Accelerators:
  - *Accelerators* are keyboard shortcuts that let you activate menu items from the keyboard.

- Decorators:
  - Menus and menu items can be decorated with another (mostly graphic) node, just like labels or buttons.

UI Controls

```
Menu fileMenu = new Menu("File");
MenuItem newItem = new MenuItem("New", new ImageView(...));
newItem.setAccelerator(KeyCombination.valueOf("Ctrl+N"));

MenuItem openItem = new MenuItem("Open", new ImageView(...));
openItem.setAccelerator(KeyCombination.valueOf("Alt+O"));

MenuItem saveItem = new CheckMenuItem("Save",
                                      new ImageView(...));
saveItem.setAccelerator(KeyCombination.valueOf("Shortcut+S"));

MenuItem exitItem = new MenuItem("Exit");
fileMenu.getItems().addAll(newItem, openItem, saveItem,
                           new SeparatorMenuItem(), exitItem);

MenuItem aboutItem = new MenuItem("About");
Menu helpMenu = new Menu("Help",
                         new ImageView(...), aboutItem);

MenuBar menuBar = new MenuBar(fileMenu, helpMenu);
VBox pane = new VBox(menuBar);
Scene scene = new Scene(pane, 190, 160);
```





UI Controls

- Key Combinations:
  - The *Shortcut* modifier is used to represent the key which is commonly used in keyboard shortcuts on the host platform (e.g. "Ctrl" on Windows and "meta" (command key) on Mac).

- Events:
  - When activated, menu items generate Action events. To handle these events, register appropriate event handlers. For example:

```
exitItem.setOnAction(e -> Platform.exit());
```

Outline

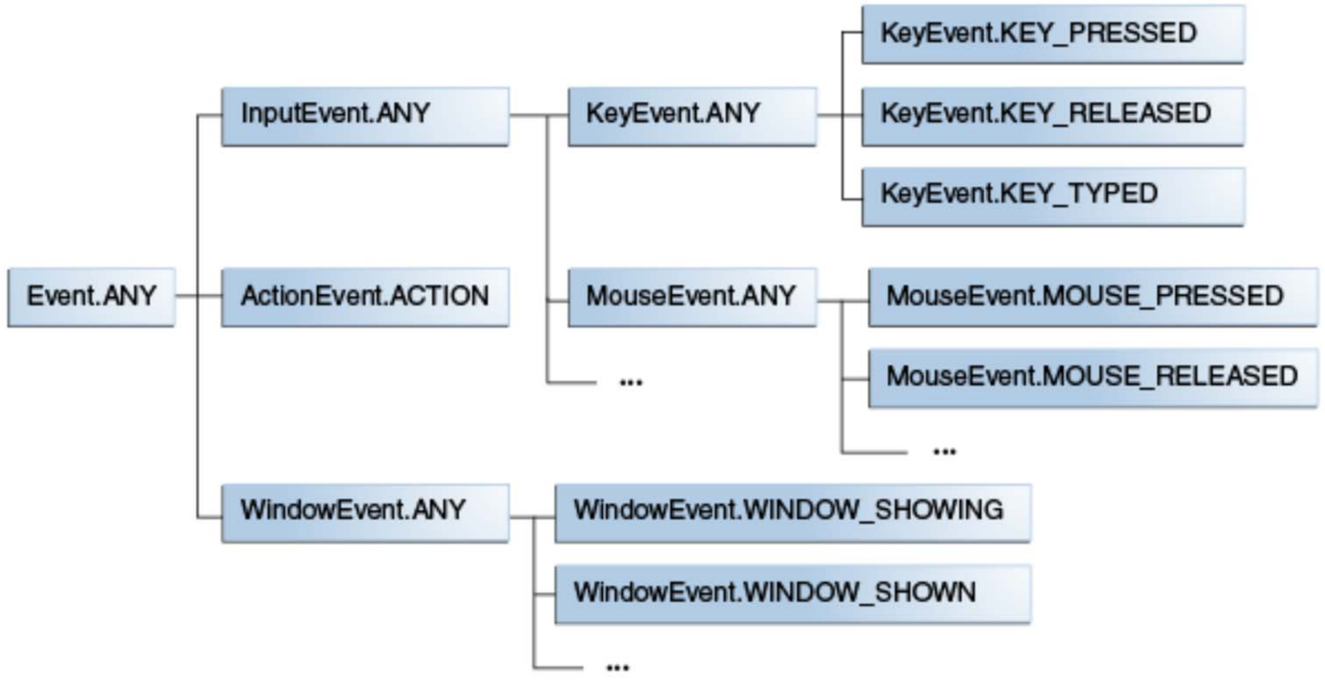
- Layout Management
- UI Controls
- Events
- Properties and Bindings
  - The JavaFX Properties
  - Bindings
  - Listeners
- Model-View-Controller (MVC)

Events

- Event:
  - An event is an object that is generated when something of interest for the application happens with one of the UI components, such as a mouse action or a key being pressed. For example:
    - A mouse being moved
    - A key being pressed
    - A window being exposed
  - In JavaFX, an event is an instance of the `javafx.event.Event` class or a subclass threreof.
    - JavaFX provides several event classes, including `ActionEvent`, `MouseEvent`, `KeyEvent` and others
  - Every event includes a *type*, a *source*, and a *target*

Events

- Event Type:
  - The event type is an instance of the `javafx.event.EventType` class. It allows the events of a single event class to be classified, so that they can be handled properly.



Events

- Event Source:
  - The event source specifies for an event handler the object on which that handler has been registered and which sent the event to it.

- Event Target:
  - The target of an event is the node on which the action occurred. It can be an instance of any class that implements the `javafx.event.EventTarget` interface

Events

- Event Delivery Process (4 steps):
  - Target Selection
  - Route Construction
  - Event Capturing Phase
  - Event Bubbling Phase

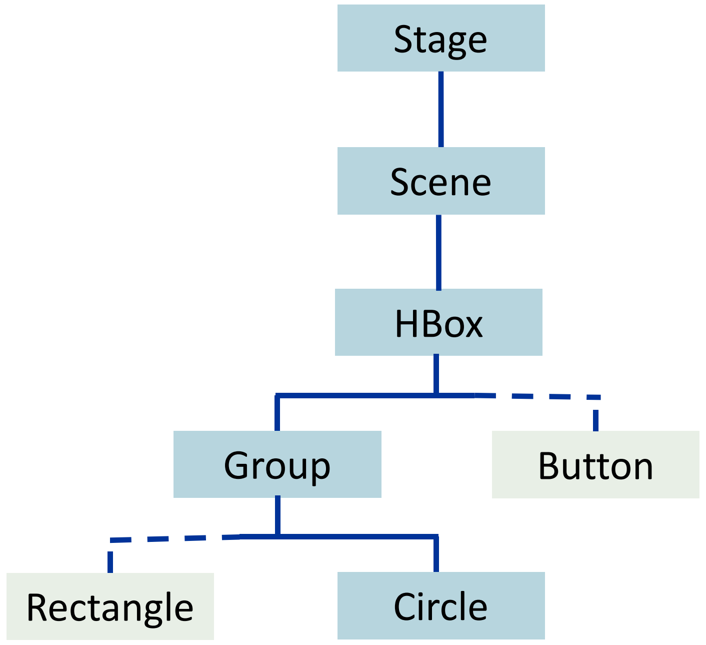
Events

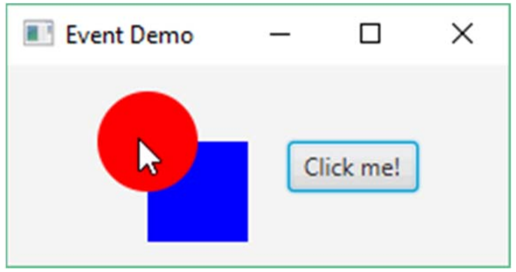
- Target Selection
  - The system determines which node is the target based on internal rules. For example:
    - For key events, the target is the node that has the focus
    - For mouse events, the target is the node at the location of the cursor

- Route Construction
  - The initial event route is determined by the *event dispatch chain* that was created in the implementation of the `buildEventDispatchChain` method of the selected event target

Events

- Example:
  - Initial route construction from the stage to the target node, when the mouse is clicked on the red circle.





```
Rectangle rect = new Rectangle(50, 50);
Circle circle = new Circle(25);
...
Group group = new Group(rect, circle);
HBox pane = new HBox(20, group, button);
Scene scene = new Scene(pane, 250, 100);
stage.setScene(scene);
```

Events

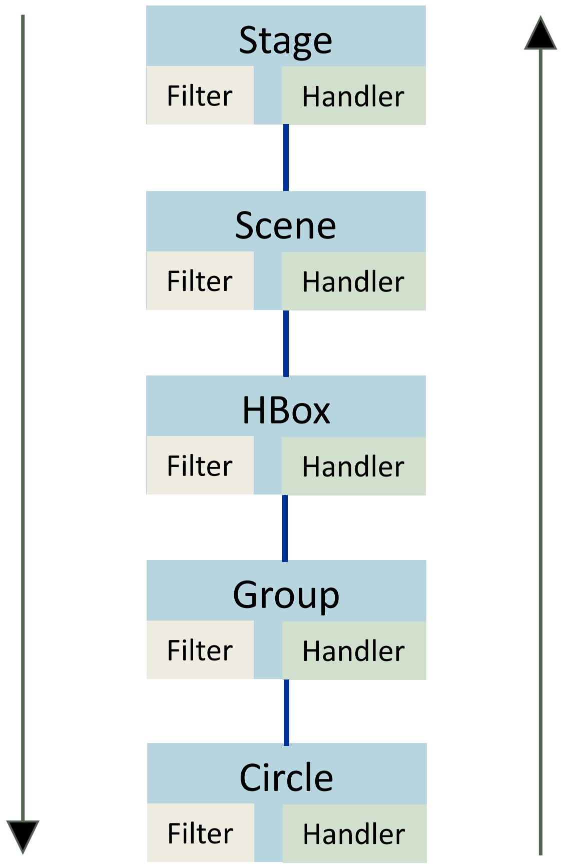
- Event Capturing Phase:
  - In the event capturing phase, the event is dispatched by the root node of the application and passed down the event dispatch chain to the target node (e.g.: from the Stage to the Circle node)
  - If any node in the chain has an *event filter* registered for the type of event that occurred, that filter is called.

- Event Bubbling Phase:
  - After the event target is reached and all registered filters have processed the event, the event returns along the dispatch chain from the target to the root node (e.g. from the Circle node to the Stage).
  - If any node in the chain has an *event handler* registered for the type of event encountered, that handler is called.

Events

- Event propagation:

Event **Capturing** Phase
→ Event **Filters** are called



Event **Bubbling** Phase
→ Event **Handlers** are called

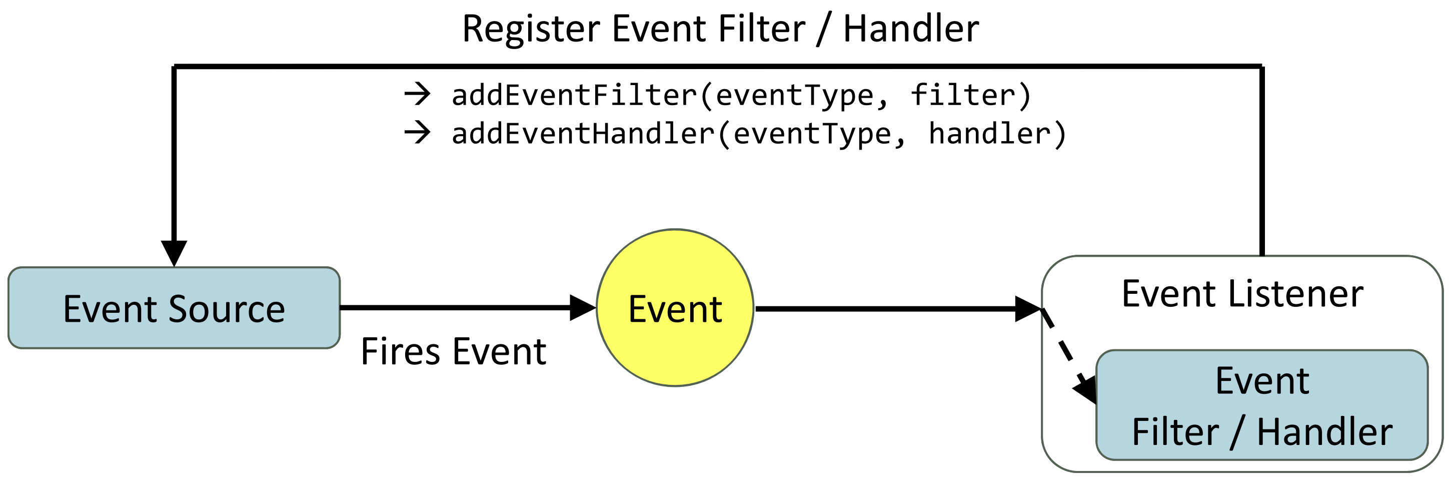
Events

- Event Handling:
  - Event handling is provided by event filters and event handlers, which are both implementations of the EventHandler interface.
  - Das EventHandler interface is a generic functional interface with a single method:

```
public void handle (T event)
```

  - In order for an event to be delivered to an event filter or handler, that filter or handler must have been registered with the event source
    - To register an event filter, resp. handler, use the method addEventFilter , resp. addEventHandler
    - To unregister them, use removeEventFilter , resp. removeEventHandler
    - All these methods accept two arguments: the event type and an event handler

Events


Register Event Filter / Handler
→ addEventFilter(eventType, filter)
→ addEventHandler(eventType, handler)
Event Source
Fires Event
Event
Event Listener
Event Filter / Handler

- Event filters or handlers can be implemented as:
  - an (outer) class
  - an inner class (anonymous or not)
  - a lambda expression

Events

- Implementation as an (outer) class:

```
class CircleHandler implements EventHandler<MouseEvent> {
  public void handle(MouseEvent e) {
    System.out.println("I was clicked!");
  }
}
```

  - In the main application class, register this handler as follows:

```
public class EventDemo extends Application {
  public void start(Stage stage) {
    Circle circle = new Circle(...);
    EventHandler<MouseEvent> handler = new CircleHandler();
    circle.addEventHandler(MouseEvent.MOUSE_PRESSED, handler);
    ...
  }
}
```

Events

- Implementation as an local inner class (in the start method):

```
public void start(Stage stage) {

  Circle circle = new Circle(...);
  // The event handler class is declared as a local class
  class CircleHandler implements EventHandler<MouseEvent> {
    public void handle(MouseEvent e) {
        System.out.println("I was clicked!");
    }
  }
  EventHandler<MouseEvent> handler = new CircleHandler();
  circle.addEventHandler(MouseEvent.MOUSE_PRESSED, handler);
  ...
}
```

Events

- Implementation as an anonymous class…

```
...
circle.addEventHandler(
   MouseEvent.MOUSE_PRESSED, new EventHandler<MouseEvent>() {
     public void handle(MouseEvent e) {
       System.out.println("I was clicked!");
   }
});
...
```

- …or a lambda expression

```
...
circle.addEventHandler(MouseEvent.MOUSE_PRESSED,
                       e -> System.out.println("I was clicked!"));
...
```

Events

- Convenience methods
  - If you only need to register a single event handler for a given event type, you will achieve the same result by using one of the convenience methods provided by JavaFX:
    - setOnAction
    - setOnMouseClicked
    - setOnMouseDragged
    - setOnKeyPressed
    - ...
  - Example:

```
...
circle.setOnMousePressed(e -> System.out.println("I was clicked!"));
...
```

Events

- Consuming an Event
  - An event can be consumed at any point in the chain by calling the consume method. This method signals that processing of the event is complete and traversal of the event dispatch chain ends.

```
circle.addEventHandler(MouseEvent.MOUSE_PRESSED,
    e -> { System.out.println("I was clicked!");
    // Event will not execute the bubbling phase
    e.consume();
});
```

  - Note that the default handlers for the JavaFX UI controls typically consume most of the input events.

Events

- Please note:
  - To register a filter, just replace addEventHandler with addEventFilter, e.g.:

```
      circle.addEventFilter(MouseEvent.ANY, handler);
```

  - A single filter/handler can be used for more than one event type.
  - A node can register more than one filter/handler. The order in which each filter/handler is called is based on the hierarchy of event types.
    - Filters/Handlers for a specific event type are executed before filters/handlers for generic event types
    - The order in which two filters/handlers at the same level are executed is not specified. Exception: the handlers registered by a convenience methods are executed last

Events

- Example:

```
circle.addEventHandler(MouseEvent.ANY,
    e -> System.out.println("Handler for ANY mouse event "
                           + "is coming last.")
);
circle.addEventHandler(MouseEvent.MOUSE_PRESSED,
    e -> System.out.println("Handler for MOUSE_PRESSED "
                           + "event is coming first.")
);
circle.setOnMousePressed(
    e -> System.out.println("Handler for convenience MousePressed "
                           + "method is coming at second.")
);
```

Events

- Some methods of an `Event` object are:
  - `getTarget`: returns the target node, i.e. the node where action occurred and the end end in the event dispatch chain
  - `getSource`: returns the source node, i.e. the node on which the filter/handler has been registered. (The source changes as the event is passed along the chain).
  - `getEventType: returns the event type`
  - `consume`: marks an event as consumed, terminating the event processing

- According to the event type, specific methods provide a lot of information about the event, e.g.:
  - the coordinates of the mouse
  - the code of keypad keys;
  - etc.

Outline

- Layout Management
- UI Controls
- Events
- Properties and Bindings
  - The JavaFX Properties
  - Bindings
  - Listeners
- Model-View-Controller (MVC)

The JavaFX Properties

- For many years, Java developed the concept of *JavaBeans Architecture* to represent and manipulate the property of an object in a standard way.

- The JavaFX property support is based on the JavaBeans model and consists of both an API and a design pattern. In this model:
  - Data ("properties") are encapsulated in an object("Bean")
  - Properties can be observed (are `Observable`)
  - Properties can be bind together (→ Bindings)
  - The architecture is based on simple name conventions

- The JavaFX properties and binding methods are mainly stored in the packages:

```
        javafx.beans                javafx.beans.binding
        javafx.beans.property       javafx.beans.value
```

The JavaFX Properties

- The package javafx.beans.property contains abstract classes for properties that represent primitive data types, strings, and other objects. Each class has a default implementation, whose name begins with "Simple":

```
    BooleanProperty bProp = new SimpleBooleanProperty(this, "b", true);
    IntegerProperty iProp = new SimpleIntegerProperty(this, "i", 3);
    DoubleProperty dProp = new SimpleDoubleProperty(this, "d", 1.5);
    ... (analog for other primitive types) ...
    StringProperty sProp = new SimpleStringProperty(this, "s", "Hallo");
```

- The constructor arguments are:
  - The object ("bean") that contains the property, e.g. this
  - The property name, e.g. "b"
  - The property initial value, e.g. true

The JavaFX Properties

- In an ObjectProperty instance, you can wrap an arbitrary object. For example:

```
ObjectProperty<Image>
    objProp = new SimpleObjectProperty<>(this, "img", new Image(...));
```

- Each property implementation class also provides convenience constructors. For example:

```
BooleanProperty bProp = new SimpleBooleanProperty();
BooleanProperty bProp = new SimpleBooleanProperty(true);
BooleanProperty bProp = new SimpleBooleanProperty(this, "b");
BooleanProperty bProp = new SimpleBooleanProperty(this, "b", true);
```

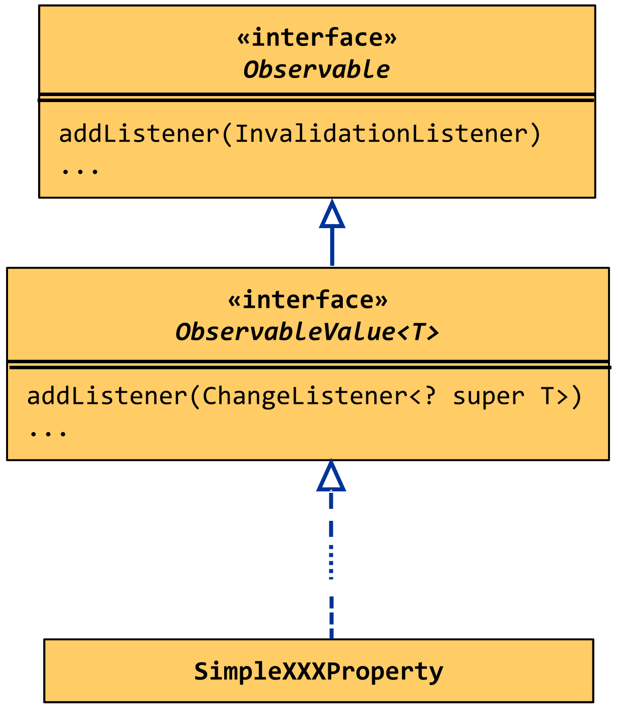
The JavaFX Properties

- The Property Pattern applied to a `BankAccount` class:

```
public class BankAccount {
  private StringProperty owner = new SimpleStringProperty();
  private DoubleProperty balance = new SimpleDoubleProperty();

  public final void setOwner(String value) { owner.set(value); }
  public final String getOwner() { return owner.get(); }
  public final StringProperty ownerProperty() { return owner; }

  public final void setBalance(double value) { balance.set(value); }
  public final double getBalance() { return balance.get(); }
  public final DoubleProperty balanceProperty() { return balance; }
}
```

The JavaFX Properties

- The property classes implement the `ObservableValue` and `Observable` interfaces, allowing you to register two *event listeners* on a property, using the corresponding `addListener` registration methods:
  - An `InvalidationListener`, which is an `Observable` listener
  - A `ChangeListener`, which is an `ObservableValue` listener



The JavaFX Properties

- The `InvalidationListener` has a single method `invalidated` that is notified when the current property value is no longer valid (more explanations: see section *Bindings).*

```
public interface InvalidationListener {
  public void invalidated(Observable obs);
}
```

- The `ChangeListener` has a single method `changed` that is notified when the property value changes. The method parameters contain the previous and the new values.

```
public interface ChangeListener<T> {
  public void changed(ObservableValue<? Extends T> obsVal,
                      T oldVal, T newVal);
}
```

The JavaFX Properties

- Using lambda expressions make the code simpler!

```
BankAccount acc = new BankAccount();
acc.setOwner("Harry");
acc.setBalance(1000);
// Register an InvalidationListener and a ChangeListener
acc.balanceProperty().addListener(obs ->
                          System.out.println("Invalidated - " + obs));
acc.balanceProperty().addListener((obsV, oldV, newV) ->
   System.out.println("Changed - " + obsV + ": " + oldV + "->" + newV);

acc.setBalance(acc.getBalance() + 500); // Notify both listeners
```

- Output:

```
Invalidated - DoubleProperty [value: 1500.0]
Changed - DoubleProperty [value: 1500.0]: 1000.0->1500.0
```

The JavaFX Properties

- Note that you can also define read-only properties by using corresponding wrapper classes:

```
public class Person {
  private ReadOnlyStringWrapper name;  // For read-only property
  ...
  public Person(String n) {
    name = new ReadOnlyStringWrapper(n);
  }

  // No setName method
  public final String getName() { return name.get(); }
  public final ReadOnlyStringProperty nameProperty() {
    return name.getReadOnlyProperty();
  }
  ...
}
```

Outline

- Layout Management
- UI Controls
- Events
- Properties and Bindings
  - The JavaFX Properties
  - Bindings
  - Listeners
- Model-View-Controller (MVC)

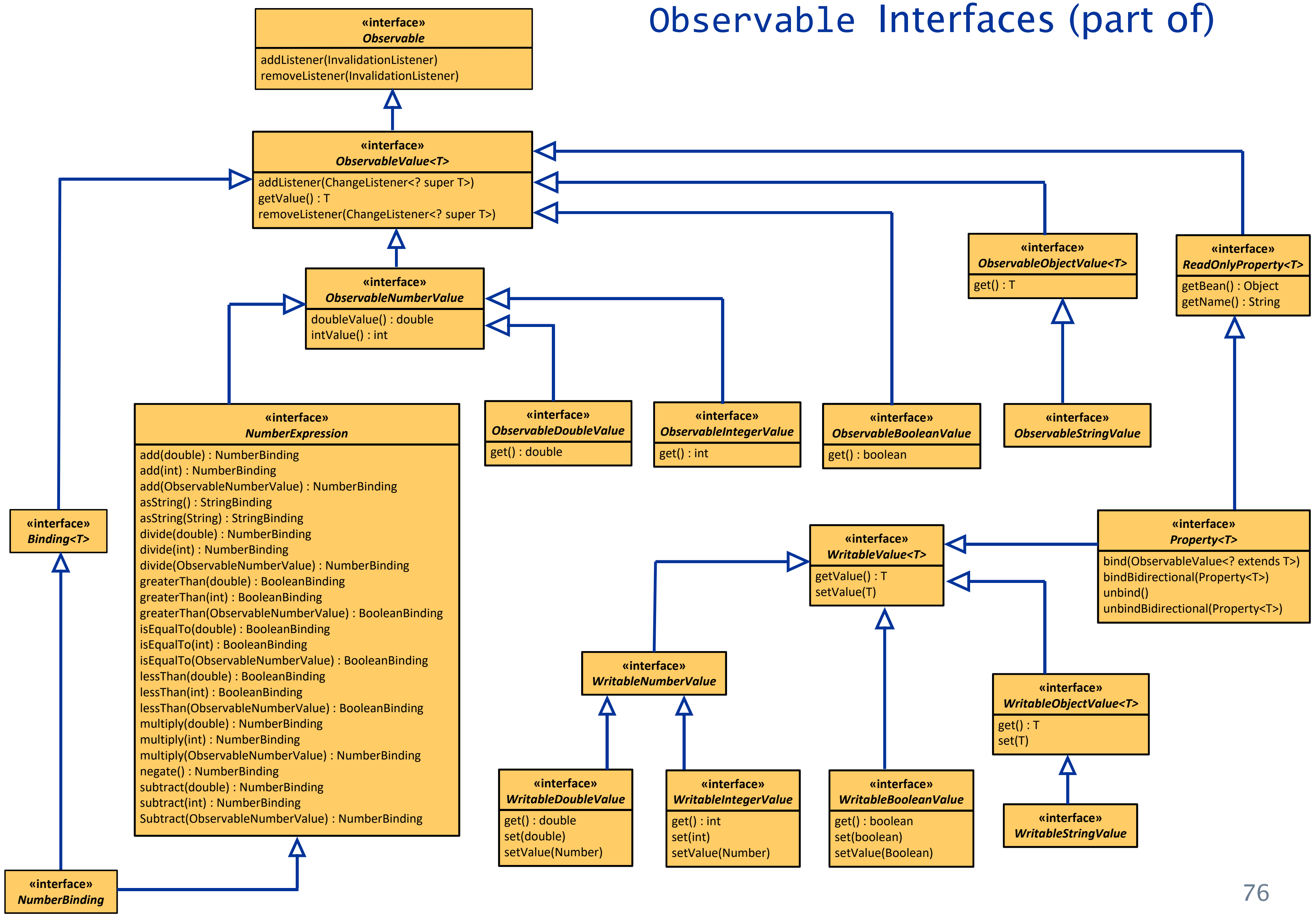
Bindings

- For this section, it can be helpful to have an overview of the `Observable` interface hierarchy and of some of its implementing classes.

- The following diagrams are not exhaustive. Some interfaces, as well as classes or methods are missing. These diagrams only aim at giving a superficial understanding of the overall structure of the interfaces and classes involved in the Property and Binding API.

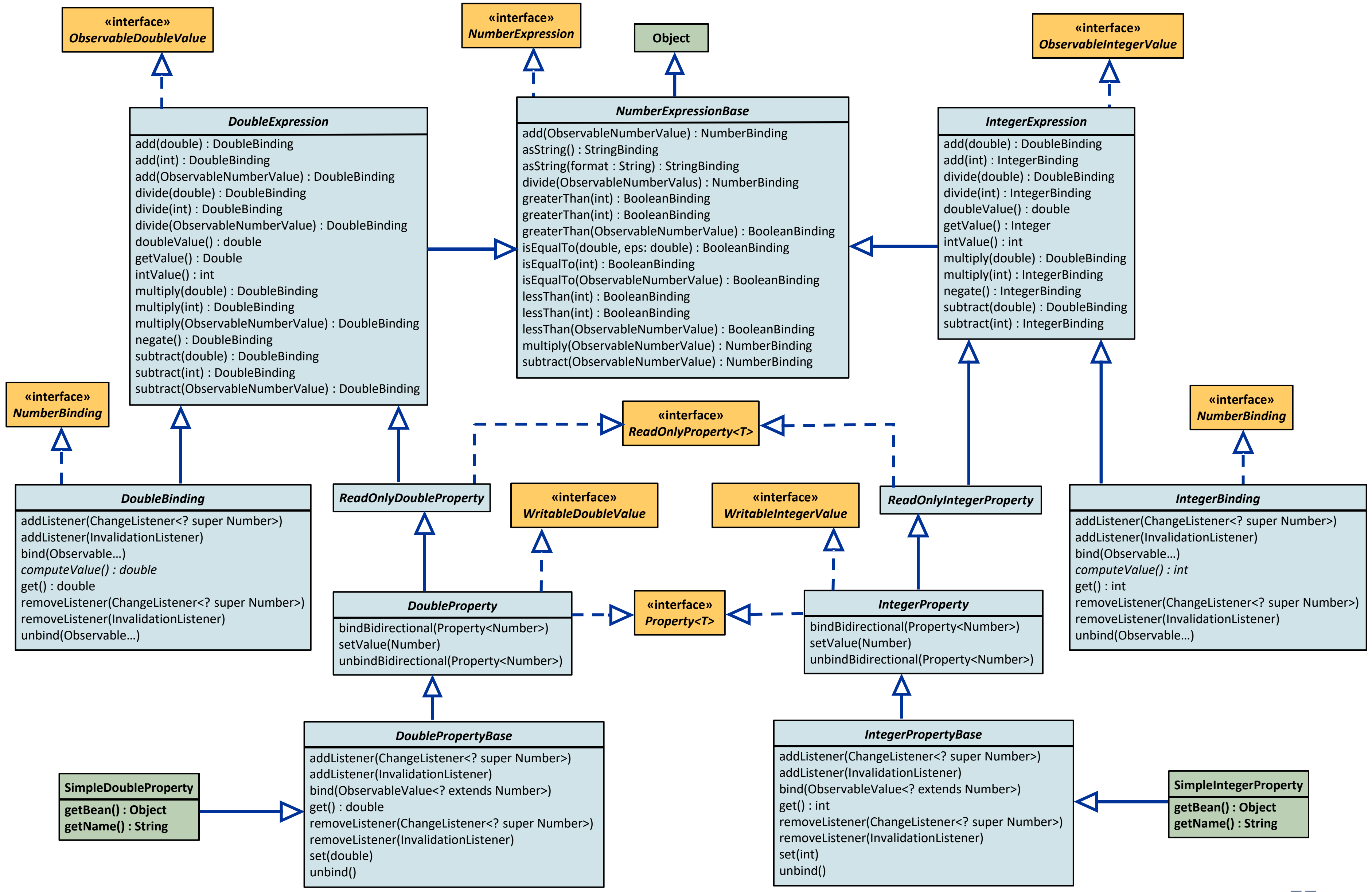
- For an in-depth understanding, look at the API documentation (package `javafx.beans` and sub-packages).

Observable Interfaces (part of)



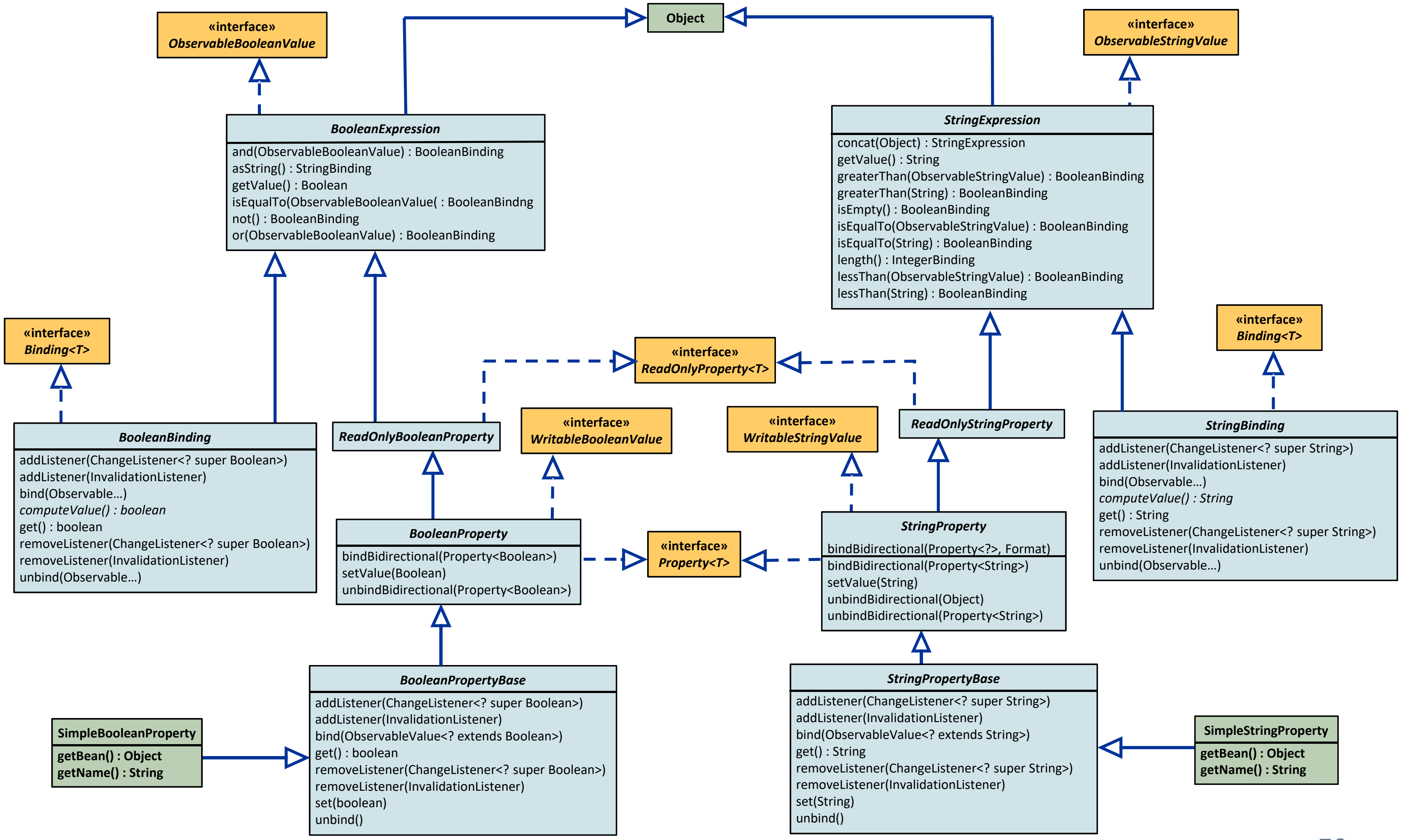
«interface» Observable
addListener(InvalidationListener)
removeListener(InvalidationListener)
«interface» ObservableValue<T>
addListener(ChangeListener<? super T>)
getValue() : T
removeListener(ChangeListener<? super T>)
«interface» ObservableNumberValue
doubleValue() : double
intValue() : int
«interface» ObservableObjectValue<T>
get() : T
«interface» ReadOnlyProperty<T>
getBean() : Object
getName() : String
«interface» NumberExpression
add(double) : NumberBinding
add(int) : NumberBinding
add(ObservableNumberValue) : NumberBinding
asString() : StringBinding
asString(String) : StringBinding
divide(double) : NumberBinding
divide(int) : NumberBinding
divide(ObservableNumberValue) : NumberBinding
greaterThan(double) : BooleanBinding
greaterThan(int) : BooleanBinding
greaterThan(ObservableNumberValue) : BooleanBinding
isEqualTo(double) : BooleanBinding
isEqualTo(int) : BooleanBinding
isEqualTo(ObservableNumberValue) : BooleanBinding
lessThan(double) : BooleanBinding
lessThan(int) : BooleanBinding
lessThan(ObservableNumberValue) : BooleanBinding
multiply(double) : NumberBinding
multiply(int) : NumberBinding
multiply(ObservableNumberValue) : NumberBinding
negate() : NumberBinding
subtract(double) : NumberBinding
subtract(int) : NumberBinding
Subtract(ObservableNumberValue) : NumberBinding
«interface» ObservableDoubleValue
get() : double
«interface» ObservableIntegerValue
get() : int
«interface» ObservableBooleanValue
get() : boolean
«interface» ObservableStringValue
«interface» Binding<T>
«interface» NumberBinding
«interface» WritableValue<T>
getValue() : T
setValue(T)
«interface» Property<T>
bind(ObservableValue<? extends T>)
bindBidirectional(Property<T>)
unbind()
unbindBidirectional(Property<T>)
«interface» WritableNumberValue
«interface» WritableDoubleValue
get() : double
set(double)
setValue(Number)
«interface» WritableIntegerValue
get() : int
set(int)
setValue(Number)
«interface» WritableBooleanValue
get() : boolean
set(boolean)
setValue(Boolean)
«interface» WritableObjectValue<T>
get() : T
set(T)
«interface» WritableStringValue

Observable Classes 1 (part of)



«interface» ObservableDoubleValue
«interface» NumberExpression
Object
«interface» ObservableIntegerValue
DoubleExpression
add(double) : DoubleBinding
add(int) : DoubleBinding
add(ObservableNumberValue) : DoubleBinding
divide(double) : DoubleBinding
divide(int) : DoubleBinding
divide(ObservableNumberValue) : DoubleBinding
doubleValue() : double
getValue() : Double
intValue() : int
multiply(double) : DoubleBinding
multiply(int) : DoubleBinding
multiply(ObservableNumberValue) : DoubleBinding
negate() : DoubleBinding
subtract(double) : DoubleBinding
subtract(int) : DoubleBinding
subtract(ObservableNumberValue) : DoubleBinding
NumberExpressionBase
add(ObservableNumberValue) : NumberBinding
asString() : StringBinding
asString(format : String) : StringBinding
divide(ObservableNumberValus) : NumberBinding
greaterThan(int) : BooleanBinding
greaterThan(int) : BooleanBinding
greaterThan(ObservableNumberValue) : BooleanBinding
isEqualTo(double, eps: double) : BooleanBinding
isEqualTo(int) : BooleanBinding
isEqualTo(ObservableNumberValue) : BooleanBinding
lessThan(int) : BooleanBinding
lessThan(int) : BooleanBinding
lessThan(ObservableNumberValue) : BooleanBinding
multiply(ObservableNumberValue) : NumberBinding
subtract(ObservableNumberValue) : NumberBinding
IntegerExpression
add(double) : DoubleBinding
add(int) : IntegerBinding
divide(double) : DoubleBinding
divide(int) : IntegerBinding
doubleValue() : double
getValue() : Integer
intValue() : int
multiply(double) : DoubleBinding
multiply(int) : IntegerBinding
negate() : IntegerBinding
subtract(double) : DoubleBinding
subtract(int) : IntegerBinding
«interface» NumberBinding
«interface» ReadOnlyProperty<T>
«interface» NumberBinding
DoubleBinding
addListener(ChangeListener<? super Number>)
addListener(InvalidationListener)
bind(Observable…)
computeValue() : double
get() : double
removeListener(ChangeListener<? super Number>)
removeListener(InvalidationListener)
unbind(Observable…)
ReadOnlyDoubleProperty
«interface» WritableDoubleValue
«interface» WritableIntegerValue
ReadOnlyIntegerProperty
IntegerBinding
addListener(ChangeListener<? super Number>)
addListener(InvalidationListener)
bind(Observable…)
computeValue() : int
get() : int
removeListener(ChangeListener<? super Number>)
removeListener(InvalidationListener)
unbind(Observable…)
DoubleProperty
bindBidirectional(Property<Number>)
setValue(Number)
unbindBidirectional(Property<Number>)
«interface» Property<T>
IntegerProperty
bindBidirectional(Property<Number>)
setValue(Number)
unbindBidirectional(Property<Number>)
DoublePropertyBase
addListener(ChangeListener<? super Number>)
addListener(InvalidationListener)
bind(ObservableValue<? extends Number>)
get() : double
removeListener(ChangeListener<? super Number>)
removeListener(InvalidationListener)
set(double)
unbind()
IntegerPropertyBase
addListener(ChangeListener<? super Number>)
addListener(InvalidationListener)
bind(ObservableValue<? extends Number>)
get() : int
removeListener(ChangeListener<? super Number>)
removeListener(InvalidationListener)
set(int)
unbind()
SimpleDoubleProperty
getBean() : Object
getName() : String
SimpleIntegerProperty
getBean() : Object
getName() : String

Observable Classes 2 (part of)



Object
«interface»
ObservableBooleanValue
BooleanExpression
and(ObservableBooleanValue) : BooleanBinding
asString() : StringBinding
getValue() : Boolean
isEqualTo(ObservableBooleanValue(: BooleanBindng
not() : BooleanBinding
or(ObservableBooleanValue) : BooleanBinding
«interface»
ObservableStringValue
StringExpression
concat(Object) : StringExpression
getValue() : String
greaterThan(ObservableStringValue) : BooleanBinding
greaterThan(String) : BooleanBinding
isEmpty() : BooleanBinding
isEqualTo(ObservableStringValue) : BooleanBinding
isEqualTo(String) : BooleanBinding
length() : IntegerBinding
lessThan(ObservableStringValue) : BooleanBinding
lessThan(String) : BooleanBinding
«interface»
Binding<T>
BooleanBinding
addListener(ChangeListener<? super Boolean>)
addListener(InvalidationListener)
bind(Observable…)
computeValue() : boolean
get() : boolean
removeListener(ChangeListener<? super Boolean>)
removeListener(InvalidationListener)
unbind(Observable…)
ReadOnlyBooleanProperty
«interface»
ReadOnlyProperty<T>
«interface»
WritableBooleanValue
«interface»
WritableStringValue
ReadOnlyStringProperty
«interface»
Binding<T>
StringBinding
addListener(ChangeListener<? super String>)
addListener(InvalidationListener)
bind(Observable…)
computeValue() : String
get() : String
removeListener(ChangeListener<? super String>)
removeListener(InvalidationListener)
unbind(Observable…)
BooleanProperty
bindBidirectional(Property<Boolean>)
setValue(Boolean)
unbindBidirectional(Property<Boolean>)
«interface»
Property<T>
StringProperty
bindBidirectional(Property<?>, Format)
bindBidirectional(Property<String>)
setValue(String)
unbindBidirectional(Object)
unbindBidirectional(Property<String>)
BooleanPropertyBase
addListener(ChangeListener<? super Boolean>)
addListener(InvalidationListener)
bind(ObservableValue<? extends Boolean>)
get() : boolean
removeListener(ChangeListener<? super Boolean>)
removeListener(InvalidationListener)
set(boolean)
unbind()
SimpleBooleanProperty
getBean() : Object
getName() : String
StringPropertyBase
addListener(ChangeListener<? super String>)
addListener(InvalidationListener)
bind(ObservableValue<? extends String>)
get() : String
removeListener(ChangeListener<? super String>)
removeListener(InvalidationListener)
set(String)
unbind()
SimpleStringProperty
getBean() : Object
getName() : String

Bindings

- Very frequently, when the value of a property changes, it is necessary to react by updating the values of one or more other properties that depends on the first one.
- This could be implemented by using a `ChangeListener`.

```
BankAccount acc1 = new BankAccount();
BankAccount acc2 = new BankAccount();

acc1.ownerProperty()
  .addListener((obsVal, oldVal, newVal) -> acc2.setOwner(newVal));

acc1.setOwner("Harry"); // Set owner "Harry" by acc1 AND acc2
acc1.setOwner("Harry Potter"); // Dito for "Harry Potter"
```

Bindings

- However, this mechanism is already implemented in the API. Properties can be *bind together*, in order to provide automatic monitoring and linking of dependencies
- Methods:
  - `bind(ObservableValue<? extends T> obsVal):` creates a unidirectional binding for this property to an observable value
  - `unbind():` removes the unidirectional binding for this property
  - `bindBidirectional(Property<T> other):` creates a bidirectional binding for this property to another one
  - `unbindBidirectional(Property<T> other):` removes a unidirectional binding for this property to another one

Bindings

```
BankAccount acc1 = new BankAccount();
BankAccount acc2 = new BankAccount();

acc2.ownerProperty().bind(acc1.ownerProperty());
acc1.setOwner("Harry");
System.out.println(acc2.getOwner()); // Prints "Harry"
acc1.setOwner("Harry Potter");
System.out.println(acc2.getOwner()); // Prints "Harry Potter"
acc2.ownerProperty().unbind();

acc2.ownerProperty().bindBidirectional(acc1.ownerProperty());
acc1.setOwner("Harry");
System.out.println(acc2.getOwner()); // Prints "Harry"
acc2.setOwner("Harry Potter");
System.out.println(acc1.getOwner()); // Prints "Harry Potter"
acc2.ownerProperty().unbindBidirectional(acc1.ownerProperty());
acc1.setOwner("Harry");
System.out.println(acc2.getOwner()); // Prints "Harry Potter"
```

Bindings

- With the Binding API, it is possible to monitor automatically much more complex value dependencies together. This API defines a set of interfaces that enable objects to be notified when a value change or invalidation takes place.
  - For example, you can perform arithmetic operations with numeric property values

- According to the complexity of the dependencies, JavaFX provides a *high-level* and a *low-level* API to handle them.

- The high-level API itself consists of two parts:
  - The *Fluent-API*, which exposes methods on the various dependency objects; and
  - The `Bindings` class, which provide static factory methods instead

Bindings

- Note that the JavaFX binding and property implementations all support *lazy evaluation*, which means that when a change occurs, the value is not immediately recomputed. Recomputation happens later, if and when the value is subsequently requested.

- Example:
  - Calculate the total amount of money contained in two bank accounts (Currency: CHF) and give the result in Euros.

Bindings

- Using the Fluent-API

```
BankAccount acc1 = new BankAccount(); acc1.setBalance(1000);
BankAccount acc2 = new BankAccount(); acc2.setBalance(2000);

// Exchange rate CHF -> EUR
DoubleProperty rate = new SimpleDoubleProperty(0.90);

// Binding to compute the total in EUR (fluent API)
NumberBinding total =
    rate.multiply(acc1.balanceProperty().add(acc2.balanceProperty()));

// total is NOT computed until explicitely requested (LAZY Binding)
System.out.println(total); // Prints "DoubleBinding [invalid]"
System.out.println("Total: " + total.getValue()); // Prints "Total: 2700.0"
System.out.println(total); // Prints "DoubleBinding [value: 2700.0]"

acc1.setBalance(8000);
rate.setValue(0.91);
System.out.println("Total: " + total.getValue()); // Prints "Total: 9100.0"
```

Bindings

- Using the `Bindings` class

```
...
// Binding to compute the total in EUR (Bindings class)
NumberBinding total =
    Bindings.multiply(rate, Bindings.add(acc1.balanceProperty(),
                                         acc2.balanceProperty()));
...
```

- Other numeric operations are supported:

  `subtract`   `divide`   `negate`   `max` (*)   `min` (*)

(*) : Only in `Bindings`

Bindings

- Using the Low-level API
  - This API provides much more flexibility. It involves extending one of the binding classes (e.g. `DoubleBinding`) and overriding its `computeValue` method to return the current value of the binding.

```
...
// Binding to compute the total in EUR (low-level API)
NumberBinding total = new DoubleBinding() {// Subclass of DoubleBinding
   {
     bind(acc1.balanceProperty(), acc2.balanceProperty(), rate);
   }

   @Override protected double computeValue() {
     return rate.get() * (acc1.getBalance() + acc2.getBalance());
   }
};
...
```

Bindings

– Note that for similar binding operations, the `Bindings` class also provides `static` convenience methods. For example:
  - `DoubleBinding createDoubleBinding(Callable<Double> func, Observable... dependencies)`

```
...
// Binding to compute the total in EUR (Bindings class)
NumberBinding total = Bindings.createDoubleBinding(() ->
                  rate.get() * (acc1.getBalance() + acc2.getBalance()),
                  acc1.balanceProperty(), acc2.balanceProperty(), rate);
...
```

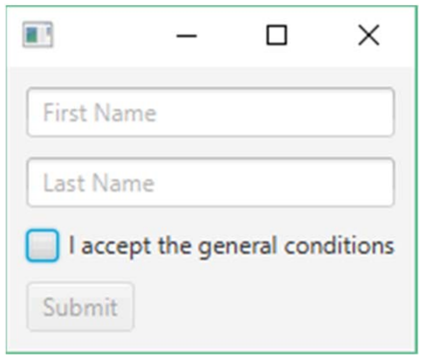
To make `double` value "`Callable`"

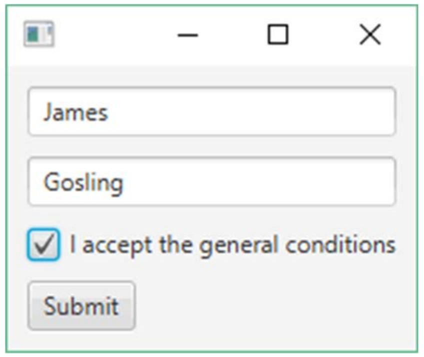
  - Other similar methods are:
    - `IntegerBinding createIntegerBinding(Callable<Integer> func, ...)`
    - `StringBinding createStringBinding(Callable<String> func, ...)`
    - `<T> ObjectBinding<T> createObjectBinding(Callable<T> func, ...)`
    - `...`

Bindings

- Numerical, string or object properties can be combined in useful logical expressions of type `BooleanBinding`.

- In the following example, the "submit" button will only be enabled if:
  - A first name and a last name (with at least 3 char.) has been entered; and
  - The check box has been selected





Bindings

```
...
TextField firstName = new TextField();
firstName.setPromptText("First Name");
TextField lastName = new TextField();
lastName.setPromptText("Last Name");
CheckBox accept = new CheckBox("I accept the general conditions");
Platform.runLater(() -> accept.requestFocus());
Button submit = new Button("Submit");
VBox rootPane = new VBox(10, firstName, lastName, accept, submit);
rootPane.setPadding(new Insets(10));

BooleanBinding nameEntered = firstName.textProperty().isNotEmpty()
   .and(lastName.textProperty().length().greaterThan(2));
submit.disableProperty()
  .bind(nameEntered.and(accept.selectedProperty()).not());

Scene scene = new Scene(rootPane);
...
```

Bindings

- Other methods that return `BooleanBindings` are:

  `lessThan, isEqualTo, isNotEqualTo, ...`

- With the `When` class, you can even select a binding according to a condition. Basically, a "`When-expression`" has the form:

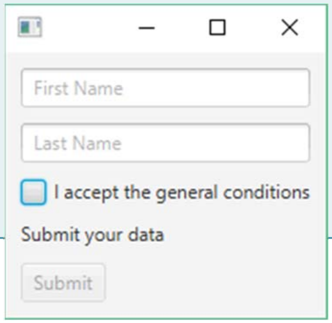
  `new When(condition).then(value1).otherwise(value2)`

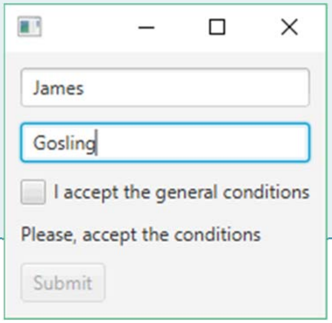
  – *condition* is an `ObservableBooleanValue` (e.g. a `BooleanProperty` or a `BooleanBinding`)
  – *`value1`* and *`value2`* have both to be of the same type (and should both be present)
    - They can be constant values or implementations of an `ObservableValue` (e.g. a `Property` or a `Binding`)
    - They determine the type of the "When-expression" (e.g. a `StringBinding`, if they are a `String` or an `ObservableStringValue`)

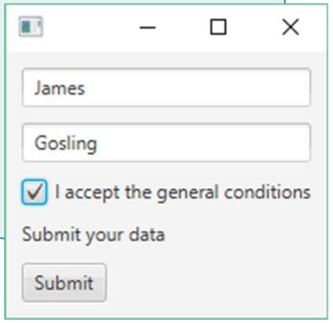
Bindings

- Example:

```
... declare two Textfields, a CheckBox and a Button
...........(see previous example) .............

Label message = new Label();
VBox rootPane = new VBox(10, firstName, lastName, accept, message, submit);
rootPane.setPadding(new Insets(10));

BooleanBinding nameEntered = firstName.textProperty().isNotEmpty()
  .and(lastName.textProperty().length().greaterThan(2));
submit.disableProperty()
  .bind(nameEntered.and(accept.selectedProperty()).not());

StringBinding helpInfo =
  new When(nameEntered.and(accept.selectedProperty().not()))
  .then("Please, accept the conditions")
  .otherwise("Submit your data");
message.textProperty().bind(helpInfo);

Scene scene = new Scene(rootPane);
...
```


First Name
Last Name
☐ I accept the general conditions
Submit your data
Submit

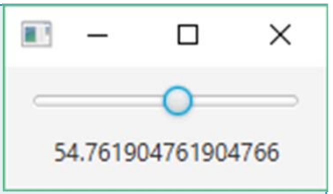

James
Gosling
☐ I accept the general conditions
Please, accept the conditions
Submit


James
Gosling
✓ I accept the general conditions
Submit your data
Submit

Bindings

- With the `asString` method, you can convert a `NumberBinding` or a `BooleanBinding` expression to a `StringBinding`:

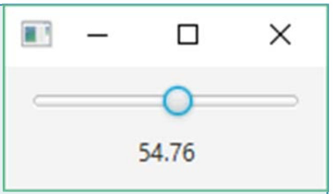
```
Slider input = new Slider(0, 100, 50);
Label display = new Label();
display.textProperty()
  .bind(input.valueProperty().asString());
...
```



54.761904761904766

- For a `NumberBinding`, you can also specify a formatting string

```
Slider input = new Slider(0, 100, 50);
Label display = new Label();
display.textProperty()
  .bind(input.valueProperty().asString("%5.2f"));
...
```



54.76

Bindings

- Sometimes, you will need to build an `ObservableValue` by concatenating a few objects together. For this purpose, the `Bindings` class provide the method:

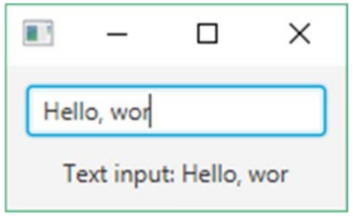
```
    public static StringExpression concat(Object... args)
```

  - This method returns a `StringExpression` (which is an `ObservableValue`) that holds the value of the concatenation of multiple objects.
  - If one of its arguments implements `ObservableValue` and the value of this `ObservableValue` changes, the change is automatically reflected in the `StringExpression`.

Bindings

- For example, use `Bindings.concat` if you need to insert a given string in a binding operation:

```
TextField input = new TextField();
Label display = new Label();
display.textProperty()
  .bind(Bindings.concat("Text input: ", input.textProperty()));

VBox rootPane = new VBox(10, input, display);
...
```



```
Hello, wor
Text input: Hello, wor
```

Bindings

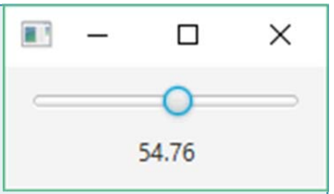
- Remember that the `Bindings` class contains a lot of convenient methods. Another example is the `format` method:

```
public static StringExpression format(String format, Object... args)
```

  - This method returns a `StringExpression` that holds the value of one or several objects formatted according to a format string.

  - You could use it to format the slider value in our previous example:

```
Slider input = new Slider(0, 100, 50);
Label display = new Label();
display.textProperty()
  .bind(Bindings.format("%5.2f", input.valueProperty()));
...
```



Outline

- Layout Management
- UI Controls
- Events
- Properties and Bindings
  - The JavaFX Properties
  - Bindings
  - Listeners
- Model-View-Controller (MVC)

Listeners

- As the property classes, the binding classes also implement the `ObservableValue` and `Observable` interfaces.

- Therefore, you can also register event listeners on a binding object. Note that:

  - The `InvalidationListener` is called when the binding becomes invalid, i.e. when the binding value *may have* changed. However, the new value will not be re-calculated, as long as it has not been explicitly requested ("lazy evaluation").

  - The `ChangeListener` is called when the binding value has changed, passing the old and the new values as parameters

Listeners

- Using the `InvalidationListener`:

```
BankAccount acc1 = new BankAccount(); acc1.setBalance(1000);
BankAccount acc2 = new BankAccount(); acc2.setBalance(2000);

// Exchange rate CHF -> EUR
DoubleProperty rate = new SimpleDoubleProperty(0.90);

// Binding to compute the total in EUR
NumberBinding total =
     rate.multiply(acc1.balanceProperty().add(acc2.balanceProperty()));
System.out.println("Total: " + total.getValue()); // Prints "Total: 2700.0"

// Register an InvalidationListener
total.addListener(obs -> System.out.println(obs));

acc1.setBalance(2000); // Listener prints "DoubleBinding [invalid]"
acc2.setBalance(1000); // No listener call
rate.setValue(0.91);   // No listener call
System.out.println("Total: " + total.getValue()); // Prints "Total: 2730.0"
...
```

Listeners

- Using the `ChangeListener`:

```
BankAccount acc1 = new BankAccount(); acc1.setBalance(1000);
BankAccount acc2 = new BankAccount(); acc2.setBalance(2000);

// Exchange rate CHF -> EUR
DoubleProperty rate = new SimpleDoubleProperty(0.90);

// Binding to compute the total in EUR
NumberBinding total =
     rate.multiply(acc1.balanceProperty().add(acc2.balanceProperty()));
System.out.println("Total: " + total.getValue()); // Prints "Total: 2700.0"

// Register an ChangeListener
total.addListener((obsVal, oldVal, newVal) ->
                   System.out.println("Value: " + oldVal + "->" + newVal));

acc1.setBalance(2000); // Listener prints "Values: 2700.0->3600.0"
acc2.setBalance(1000); // Listener prints "Values: 3600.0->2700.0"
rate.setValue(0.91);   // Listener prints "Values: 2700.0->2730.0"
System.out.println("Total: " + total.getValue()); // Prints "Total: 2730.0"
...
```

Listeners

- Which listener should I use?
  - The `ChangeListener` (`ObservableValue` listener) is more "comfortable", because it always provides the old and new values of the modified binding value. But it is activated for each modification of that value.

  - On the other side, the `InvalidationListener` (`Observable Listener`) is only called once when a binding value may have changed. It will not be activated again as long as the (new) value of that binding has not been requested. This could be important by performance issues.

Outline

MVC

- The MVC is an architectural concept designed to implement user interfaces.

- It teaches to group the functionalities of these applications in three separated categories of interconnected objects:
  - The *Model* , which maintains the data
  - The *View* , which represents of the data
  - The *Controller*, which handles the user input

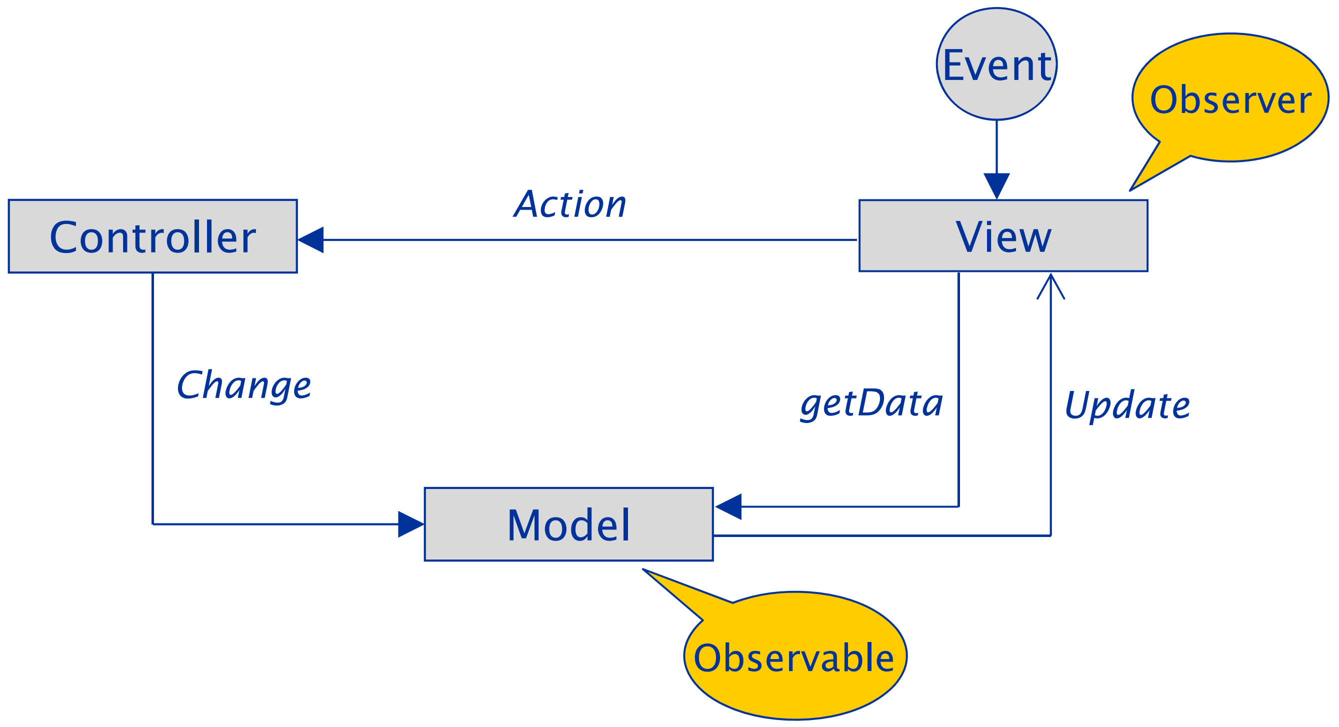
- For example, any JavaFX control typically has:
  - A *content,* such as the state of a button (pushed in or not), or the text in a text field
  - A *visual appearance* (color, size, and so on)
  - A *behavior* (reaction to events)

MVC

- Therefore, the MVC model :
  - is responsible for storing and maintaining data
  - provide corresponding methods to access and modify data
  - If necessary, notify the view, when a data change occurs (typically using the *Observer pattern)*
- ... the MCV view:
  - provides a visual representation of the model's data
  - updates itself based on the model changes
- … and the MVC controller:
  - handles input events such as mouse clicks or key strokes and decides whether to translate these events into changes in the model or the view.

MVC

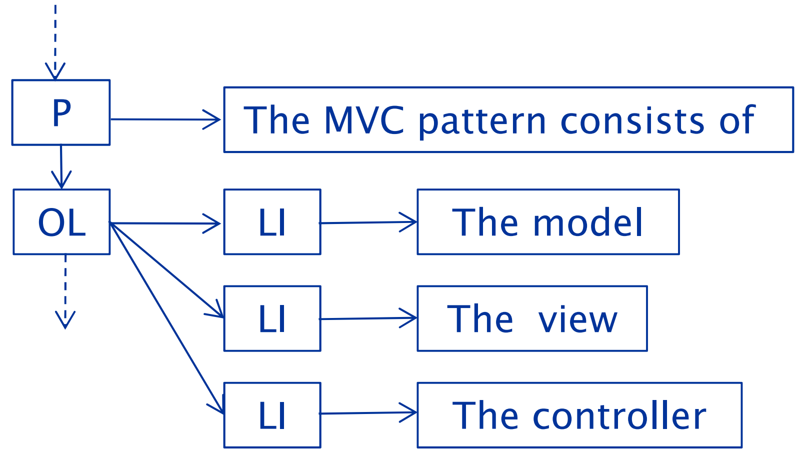
- Although the communication ways between the three MVC components may vary, a typical communication scheme is:



MVC

- One of the important advantages of the MVC pattern is that a model can have multiple views

Model:



WYSIWYG View:

The MVC pattern consists of

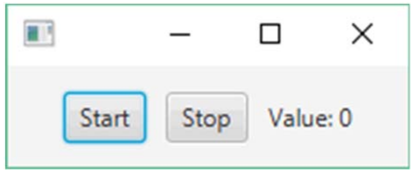
1. The model
2. The view
3. The controller

Tag View:

```
<P> The MVC pattern consists of </P>
<OL>
    <LI> The model </LI>
    <LI> The view </LI>
    <LI> The controller </LI>
```

MVC

- Example: An simple counter
  - When the start button is pressed, the counter is incremented by one each second, until the stop button is pressed



- MVC Architecture:
  - Model: the counter
    - It stores the current value and allows to increment it
  - View: the GUI
  - Controller: a timer
    - It starts or stops the counter according to the button commands.

MVC

- There are many ways to implement the MVC architecture in Java. We show here a few examples that use the JavaFX properties concept

- Implementation 1: using a `ChangeListener`

  - In this implementation, the view registers a `ChangeListener` with the model

  - When the model changes, the listener is notified and transmits the changed data with the notification ("push-variant")

MVC

– The Counter Model

```
public class CounterModel {
  private ReadOnlyIntegerWrapper value;

  public CounterModel() {
    value = new ReadOnlyIntegerWrapper(0);
  }

  public int getValue() {
    return value.get();
  }

  public void inc() {
    value.setValue(value.get() + 1);
  }

  public void addObserver(ChangeListener<Number> li) {
    value.getReadOnlyProperty().addListener(li);
  }
}
```

The value property is read-only for the view.

Allows the view to register a listener with the counterValue property

MVC

– The Counter View (GUI)

```
public class CounterView extends Stage {

  public CounterView(CounterModel model, CounterController controller) {
    Label label = new Label("Value: " + model.getValue());
    Button start = new Button("Start");
    start.setOnAction(e -> controller.start());

    ... // Build the GUI

    model.addObserver((obsVal, oldVal, newVal) ->
                      label.setText("Value: " + newVal));
  }
}
```

Gets model and controller references

Activates the Controller

Registers a ChangeListener with the model

MVC

- The Counter Controller:

```
public class CounterController {
  private AnimationTimer timer;

  public CounterController(CounterModel model) {
    timer = new AnimationTimer() {
    private long currentTime;
    public void handle(long now) {
      if (now > currentTime + 1_000_000_000) {  // One second elapsed
        currentTime = now;
        model.inc();
      }}};
  }

  public void start() { timer.start(); }

  public void stop() { timer.stop(); }
}
```

Gets the model reference

Changes the model

Activated by the button event handlers

MVC

– The Main Class:

```
public class MVCCounter extends Application {

  public void start(Stage stage) {
    CounterModel model = new CounterModel();
    CounterController controller = new CounterController(model);
    new CounterView(model, controller);
  }
}
```

Creates the MVC components

MVC

- Implementation 2: using an `InvalidationListener`
  - In this implementation, the view registers an `InvalidationListener` with the model
  - When the model changes, the listener is notified but does not transmit the changed data.
  - Therefore, the model must provide a method (→ `getValue`) for the view to fetch the data ("pull variant").

MVC

- Changes against implementation 1 are:
  - In CounterModel, redefine the addObserver method as:

```
public void addObserver(InvalidationListener<Number> li) { ... }
```

  - In CounterView, invoke the addObserver method as:

```
model.addObserver(obs -> label.setText("Value: " + model.getValue()));
```

  - The other classes are not modified

MVC

- Implementation 3: using property bindings
  - Changes against implementation 1 and 2 are:
    - In CounterModel, replace the addObserver method with:

```
    public ReadOnlyIntegerProperty valueProperty() {
      return value.getReadOnlyProperty();
    }
```

    - In CounterView, replace the invocation of the addObserver method with:

```
    label.textProperty().bind(Bindings.concat("Value: ",
                                 model.valueProperty().asString()));
```

    - The other classes are not modified

MVC

- Note that the MVC communication flow can also be found in different variants, such as:

```
public class ... Main Class ... {
  public ... Main method ... {
    CounterModel model = new CounterModel();
    CounterView view = new CounterView(model);
    CounterController controller = new CounterController(model, view);
  }
}
```

The controller also updates/informs the view

```
public class ... Main Class ... {
  public ... Main method ... {
    CounterModel model = new CounterModel();
    CounterView view = new CounterView();
    CounterController controller = new CounterController(model, view);
  }
}
```

The whole communication goes through the controller