GUIs: JavaFX (I)

Lecture 8 (19 April 2022)

Graphical user interfaces

why GUI-programming in OOP?

GUI = Graphical User Interface

- it is important to know how to make a GUI
- it uses/illustrates the use of the important concepts

Graphical User Interfaces in Java

- When Java was introduced, GUI classes were bundled in a library known as the Abstract Window Toolkit (AWT) [1995]
  - AWT is fine for developing simple graphical user interfaces, but not for developing comprehensive GUI projects.
- Swing: platform-independent unified look-and-feel [1997]
  - Model-View-Controller GUI framework
- JavaFX [circa 2007, open-sourced 2011]
  - desktop applications, rich internet applications
  - much better object oriented structure
  - different ways to use JavaFX
  - as a WYSIWYG editor (easy, but fixed layout)
  - as an OO library (using many important OO concepts)

we will use this

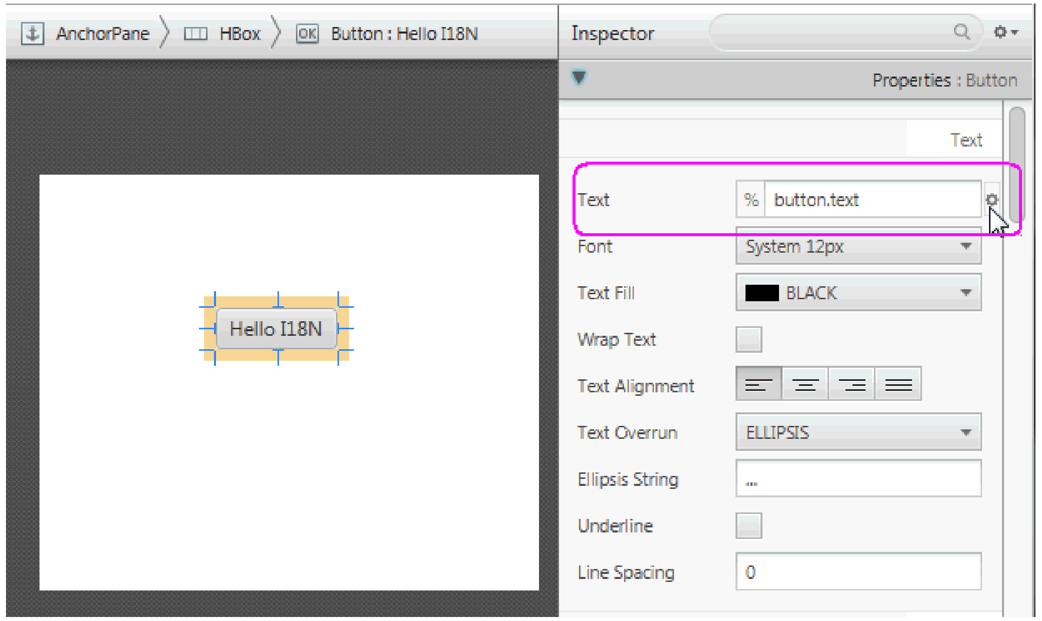
different ways of working with JavaFX

JavaFX Scene Builder

- GUI without writing code, drag-and-drop WYSIWYG interface
- standalone program integrated with NetBeans (and other IDEs)
- generates FXML markup (you have to add logic later)

JavaFX API

- use classes from the JavaFX library directly
- program the layout of the user interface
- we will use this way of working



Text	% button.text
Font	System 12px
Text Fill	BLACK
Wrap Text	
Text Alignment	
Text Overrun	ELLIPSIS
Ellipsis String	...
Underline	
Line Spacing	0

GUI – OS interaction

- OS can draw windows, buttons, menus, etc. in the look and feel of its brand
- GUI-program has to indicate what GUI objects there are and where they should be drawn
- After each window manipulation or event (mouse click, mouse movement, key click, …) things can change
  - the GUI-program has to draw (some) objects again with help of the OS
- JavaFX solution:
  - class `Application` takes care of layout and OS interaction
  - a (tree-like) data structure based on type `Node` specifies the GUI objects
  - you override the `start` of `Application` to define the `Node` tree
  - static method `launch` of `Application` makes the `Application` object and calls `start`

GUI architecture

Use the object oriented structure:

- there are classes for building the GUI components
- make instances for all actual objects in the GUI: button, menu, window, ..

Library draws objects and gives default behaviour

- pressing a button, unfolding a menu
- uses look-and-feel of host system: Windows, Mac OS, Linux, …
- user specifies specific behaviour: how to handle *events* (button pressed, menu item selected, …)

User is in control of the application

quite different from traditional console applications (Read-Eval-Print-Loop)

JavaFX program structure

```
public class MyProgram {
  public static void main(String[] args) {
    ..
  }
}
```

becomes

```
public class MyFXProgram extends Application {
  @Override
  public void start(Stage primaryStage) {
    ..
  }
  public static void main(String[] args){
    launch(args);
  }
}
```

main is always the same so we leave it out of the slides

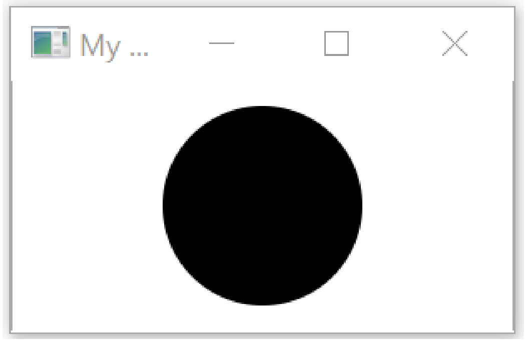
JavaFX Application life-cycle

What `launch` does:

1. creates an instance of the specified Application class
2. calls the `init` method
3. calls `start (Stage … )`
   this method is abstract in `Application`
   it must be implemented in your class
4. waits for the application to finish,
   which happens when either of the following occur:
   - the application calls `Platform.exit`
   - the last window has been closed
5. calls the `stop` method
   - e.g. close open files

first JavaFX application

```
public class MyFirstJFXClass extends Application {
  @Override
  public void start(Stage stage) {
    Circle circle  = new Circle(100, 50, 40);
    Pane root      = new Pane(circle);
    Scene scene    = new Scene(root, 200, 100);
    stage.setTitle("My First Java-FX App");
    stage.setScene(scene);
    stage.show();
  }

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



Terminology: Stage, Scene, Pane, Node

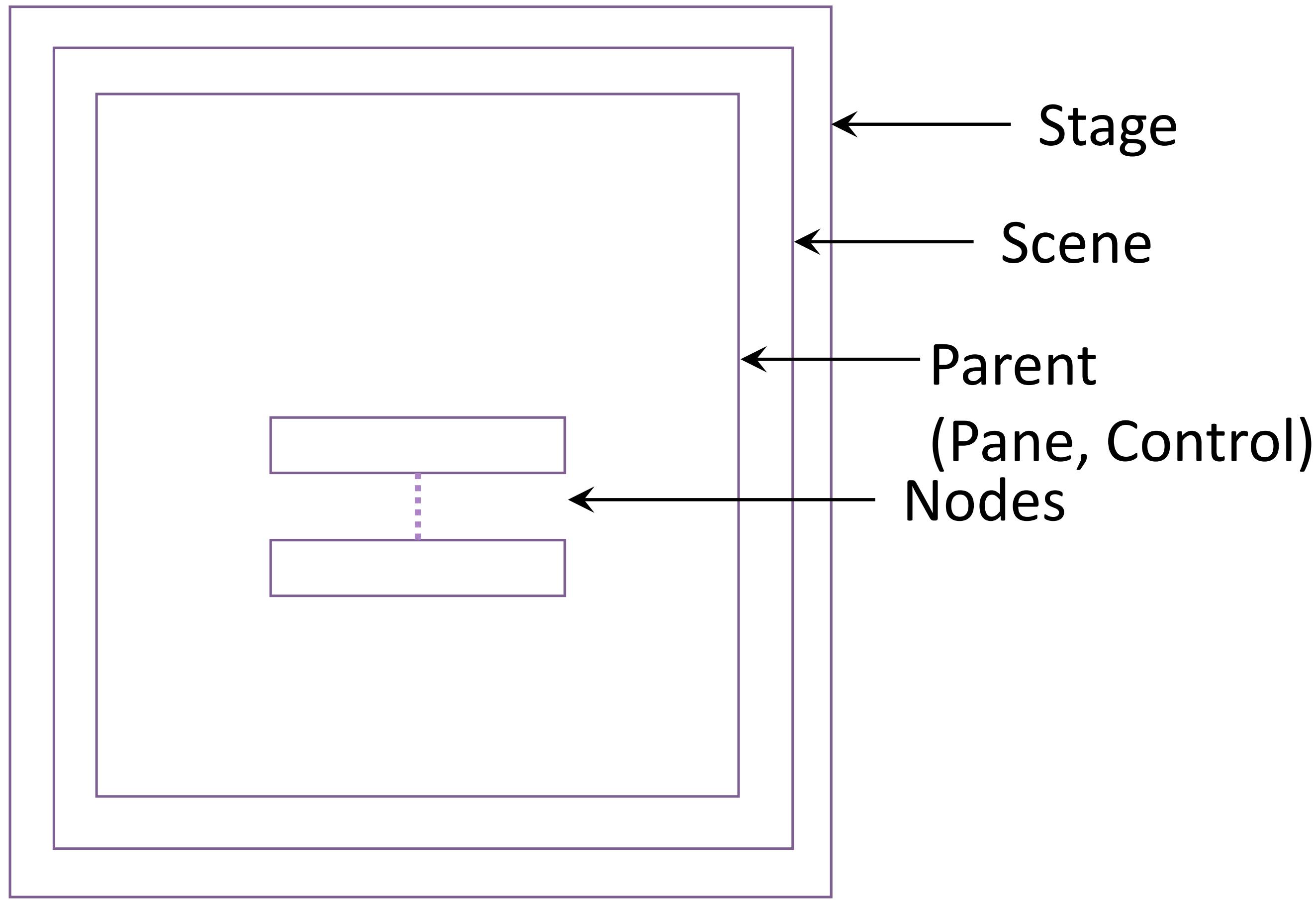
An application can have multiple stages

`Stage` has one `Scene`

`Scene` has one `Parent` (root)

`Parent`: base class for all nodes that have children in the scene graph.

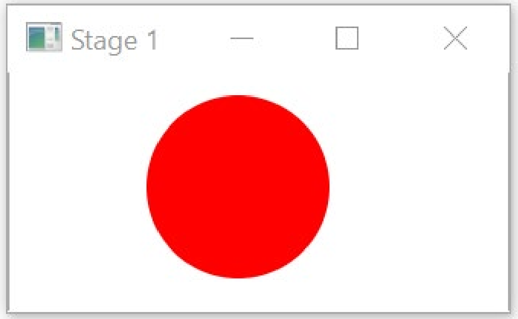
`Node`: any JavaFX component

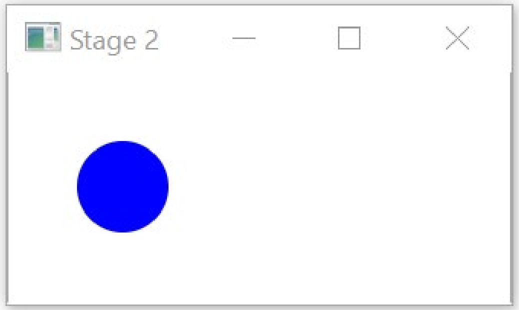


2 windows / 2 stages

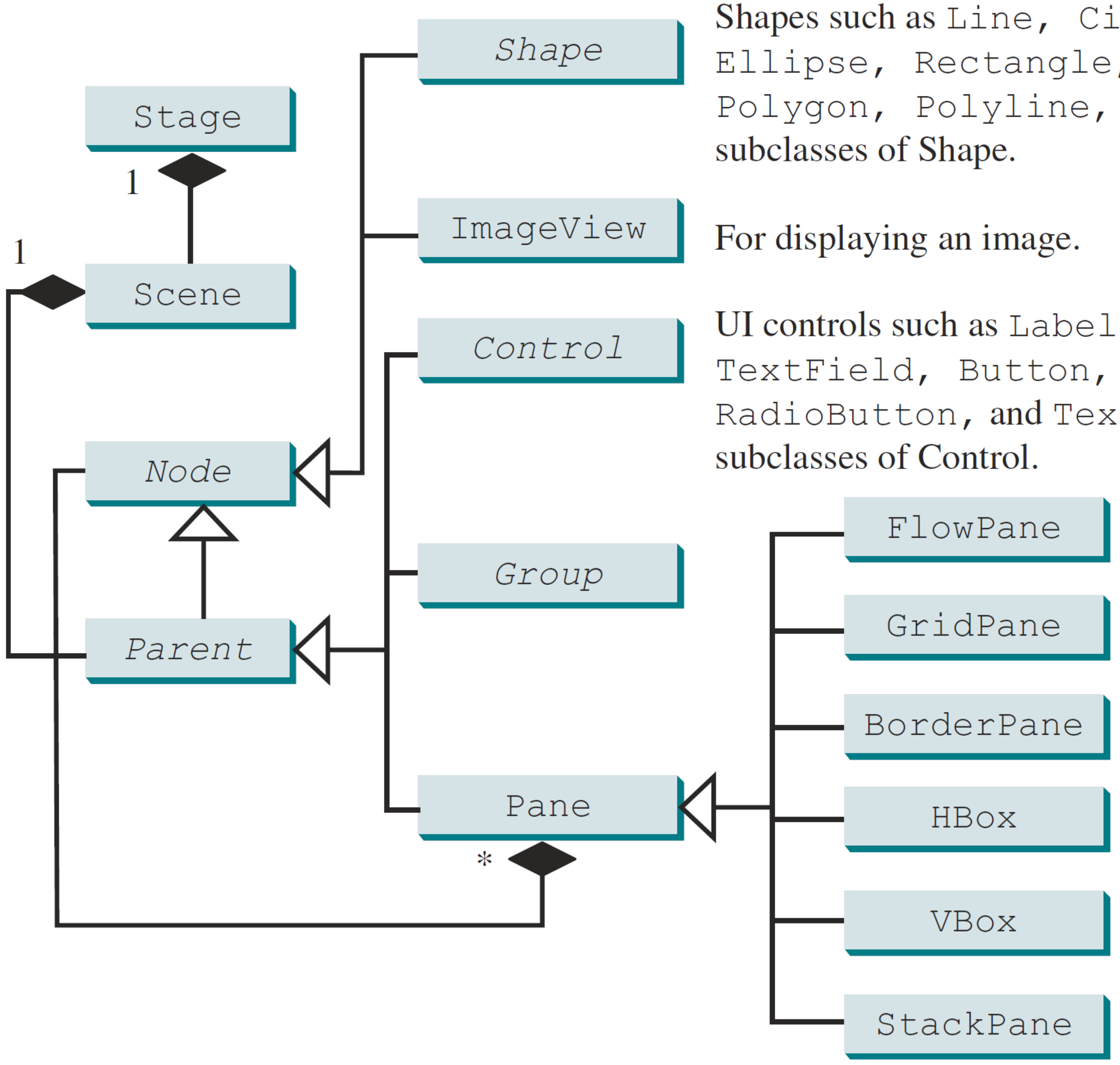
x, y, radius

```
public void start(Stage stage1) {
  Circle circle1 = new Circle(100, 50, 40);
  circle1.setFill(Color.RED);
  Scene scene = new Scene(new Pane(circle1), 200, 100);
  stage1.setTitle("Stage 1");
  stage1.setScene(scene);
  stage1.show();

  Stage stage2 = new Stage();
  Circle circle2 = new Circle(50, 50, 20);
  circle2.setFill(Color.BLUE);
  stage2.setTitle("Stage 2");
  stage2.setScene(new Scene(new Pane(circle2), 200, 100));
  stage2.show();
}
```





Stage, Scene, Pane, Node



Shapes such as `Line`, `Circle`, `Ellipse`, `Rectangle`, `Arc`, `Polygon`, `Polyline`, and `Text` are subclasses of Shape.

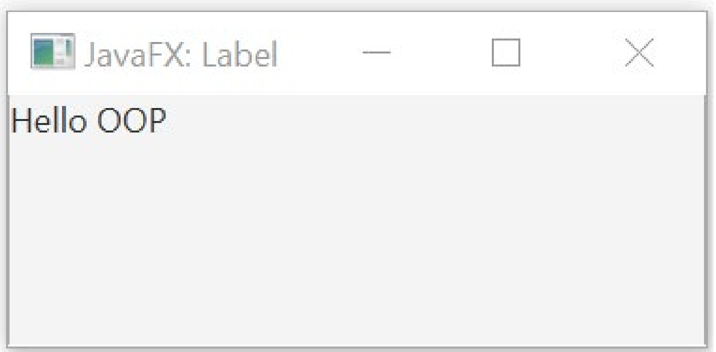
For displaying an image.

UI controls such as `Label`, `TextField`, `Button`, `CheckBox`, `RadioButton`, and `TextArea` are subclasses of Control.

Displaying text

```
public void start(Stage stage) {
    Label label = new Label("Hello OOP");
    Pane pane = new Pane(label);
    stage.setTitle("JavaFX: Label");
    stage.setScene(new Scene(pane, 200, 100));
    stage.show();
}
```

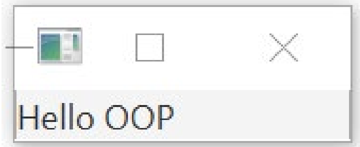
width, height



Hello OOP

Without pane, no scene size

```
public void start(Stage stage) {
    Label label = new Label("Hello OOP");
    stage.setTitle("JavaFX: Label");
    stage.setScene(new Scene(label));
    stage.show();
}
```



Hello OOP

Why do we (almost) always add a Pane?

- Our first label example has a `Pane`, the second example has no `Pane`
- `Label` is a `Control`, so no `Pane` is required.
- Any ‘real’ program has one or more `Pane` objects
    - control layout
    - set background color
    - mouse handlers
- It is customary to add a pane
    - there are various `Pane` subclasses yielding different layout of nodes

Making the GUI do stuff: Properties & Event handling

Properties

Binding Properties

- JavaFX introduces a new concept: binding property
  - binding: defines a relation between data elements (usually variables) in a program to keep them synchronized.
    - In a GUI application: used to synchronize the elements in the (data) *Model* with the corresponding UI elements of the *View*.
  - Enables a target object to be bound to a source object.
  - If the value in the source object changes, the target object is updated automatically.
  - The target object is called a *binding object* or a *binding property*.

- Properties are Java objects containing/wrapping a value
- Instead of a concrete type (`int`, `double`,…) fields often get a *`Property`* as type
  - e.g. `IntegerProperty` iso `int`
- we can bind properties to other properties

```
target.bind(source);
```

properties: getters & setters

Objects with property fields have **two** getters and **one** setter per property (convention, no hard rule)

- one getter for the value of the property, e.g. `circle.getCenterX()`
- one setter for the value of the property, e.g. `circle.setCenterX(…)`
- one getter for the Property object itself, e.g. `circle.centerXProperty()`
- no setter for the Property object – properties are mutated, not replaced
  - can be made final (or are final for JavaFX components)

properties: getters & setters (II)

```
public class SomeClassName {
  private PropertyType x;

  /** Value getter method */
  public propertyValueType getX() { ... }

  /** Value setter method */
  public void setX(propertyValueType value) { ... }

  /** Property getter method */
  public PropertyType xProperty() { ... }
}
```

```
public class Circle {
  private DoubleProperty centerX;

  /** Value getter method */
  public double getCenterX() { ... }

  /** Value setter method */
  public void setCenterX(double value) { ... }

  /** Property getter method */
  public DoubleProperty centerXProperty() { ... }
}
```

property binding demo: integers

```
private void run() {
  IntegerProperty x = new SimpleIntegerProperty(1);
  IntegerProperty y = new SimpleIntegerProperty(7);
  print(x, y);
  y.bind(x);
  print(x, y);
  y.bind(x.multiply(8).add(2));
  print(x, y);
  x.set(5);
  print(x, y);
}

private void print(IntegerProperty a, IntegerProperty b) {
  System.out.printf("%d, %d\n", a.intValue(), b.intValue());
}
```

`SimpleIntegerProperty extends IntegerProperty`

IntegerProperty is an abstract class

bind is Property method

replaces previous binding

an expression that converts the value of x

Why not: `y.bind(x*8+2);` ?

RUN

```
1, 7
1, 1
1, 10
5, 42
```

bidirectional binding demo: doubles

```
public static void run() {
  DoubleProperty d1 = new SimpleDoubleProperty(1);
  DoubleProperty d2 = new SimpleDoubleProperty(2);
  d1.bindBidirectional(d2);
  print(d1, d2);
  d1.setValue(50.1);
  print(d1, d2);
  d2.setValue(70.2);
  print(d1, d2);
}
```

RUN

```
2,000000, 2,000000
50,100000, 50,100000
70,200000, 70,200000
```

property demo: strings

```java
private void run() {
  IntegerProperty x = new SimpleIntegerProperty(1);
  IntegerProperty y = new SimpleIntegerProperty(7)
  StringProperty s  = new SimpleStringProperty();
  s.bind(Bindings.concat("X has value ", x, ", Y has value ", y));
  print(s);
  y.bind(x.multiply(8).add(2));
  print(s);
  x.set(5);
  print(s);
}
private static void print( StringProperty s )
  System.out.println(s.getValue());
}
```

concat: builds an observable String with embedded observables

Bindings is a utility class

s is being automatically updated every time x and/or y change

RUN

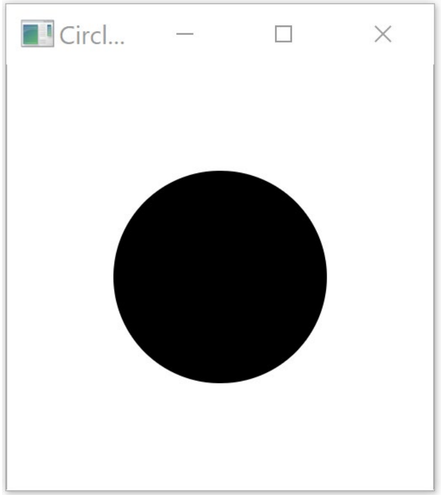
```
X has value 1, Y has value 7
X has value 1, Y has value 10
X has value 5, Y has value 42
```

example: keep circle in the middle

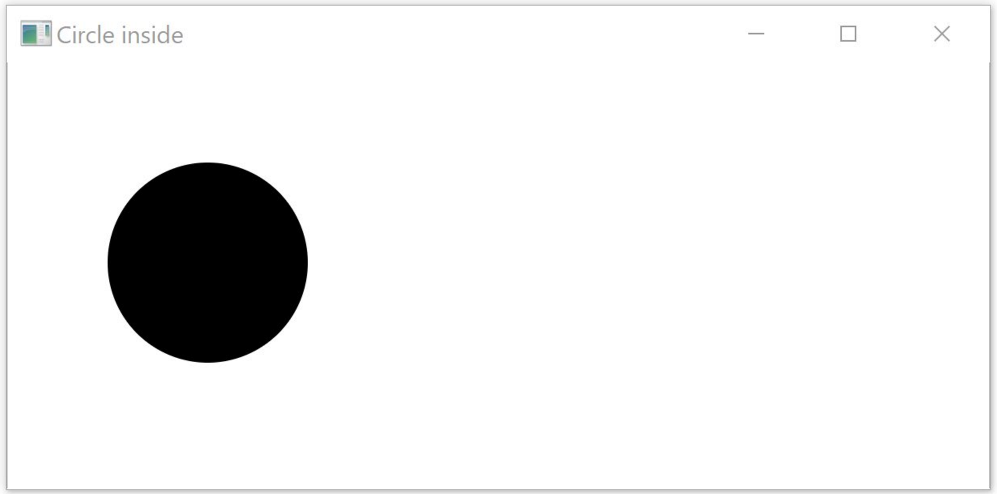
- if you resize a window the gui components will stay in place.

```
public void start(Stage stage) {
    stage.setTitle("Circle inside");
    Circle circle = new Circle(100, 100, 50);
    stage.setScene(new Scene(new Pane(circle), 200, 200));
    stage.show();
}
```

after startup



after resizing



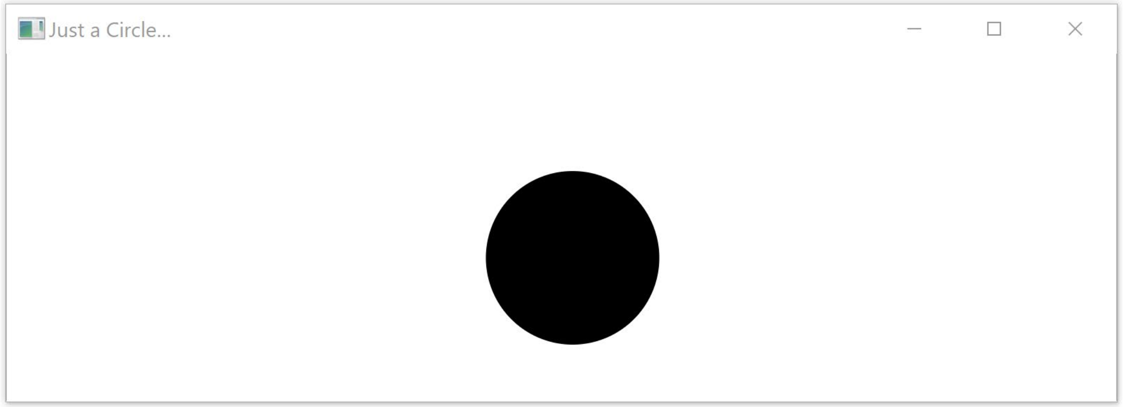
example: keep circle in the middle (II)

- How do you make sure the circle stays in the middle?
  - Answer: use property binding!
- window dimensions and circle position are Properties

```
public void start(Stage stage) {
    stage.setTitle("Just a circle…");
    Circle circle = new Circle(50);
    circle.centerXProperty().bind(stage.widthProperty().multiply(0.5));
    circle.centerYProperty().bind(stage.heightProperty().multiply(0.5));
    stage.setScene(new Scene(new Pane(circle), 200, 200));
    stage.show();
}
```

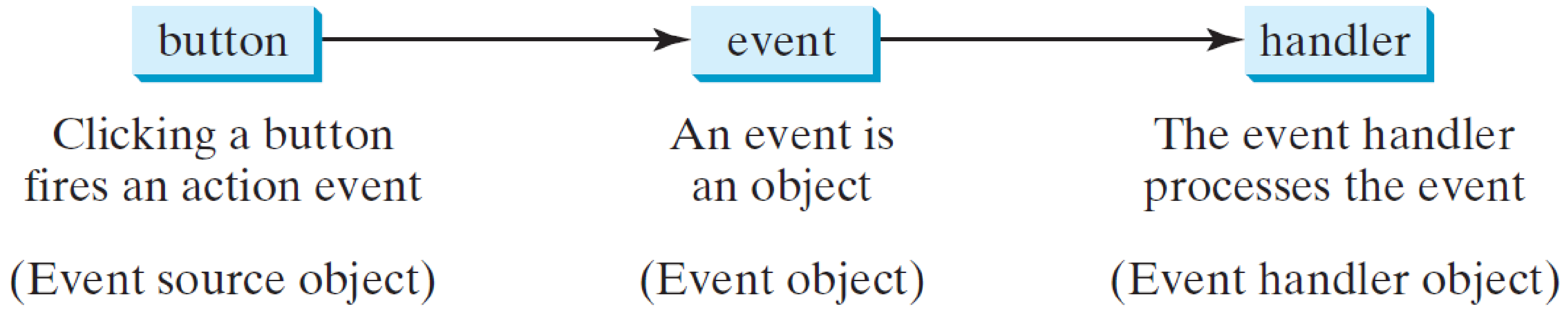
initial position is no longer needed

after resizing



Making the GUI do stuff: Event handling

handling (button) events



- JavaFX takes care of generating the event object and passing it to an appropriate handler
- We must specify the *handler*

```
interface EventHandler<T extends Event> {
  void handle(T event);
}
```

functional interface: Single Abstract Method

handler always gets the event causing the call as its argument

implementing handlers

Several ways to implement interface EventHandler

1. an separate class
2. by the class of the `this` object
3. named inner-class
4. an anonymous class
5. lambda-expression

How do we install a handler? (e.g. How do we link a handler to a button?)

- Answer: using `button.setOnAction( … )`

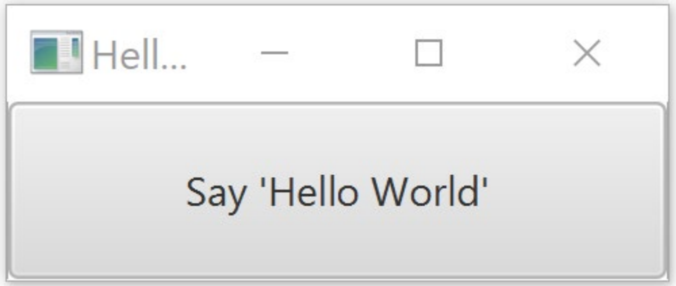
button instance

the handler goes in here

One button with anonymous class as event handler

```
public void start(Stage stage) {
   Button btn = new Button("Say 'Hello World'");
   btn.setOnAction(new EventHandler<>() {
     @Override
     public void handle(ActionEvent event) {
       System.out.println("Hello World!");
     }
     });
   Scene scene = new Scene(btn, 100, 50);
   stage.setTitle("Hello World!");
   stage.setScene(scene);
   stage.show();
 }
```

text on the button



text printed to console

title of the window

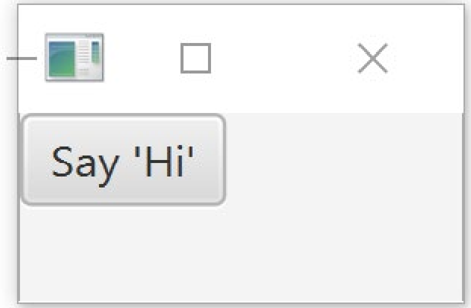
no pane: the button will fill the window completely!

button with lambda expression as event handler

```
public void start(Stage stage) {
   Button btn = new Button();
   btn.setText("Say 'Hi'");
   btn.setOnAction(e -> System.out.println("Hi"));
   Scene scene = new Scene(new Pane (btn), 100, 50);
   stage.setTitle("Hi World!");
   stage.setScene(scene);
   stage.show();
}
```

button text is set in separate call

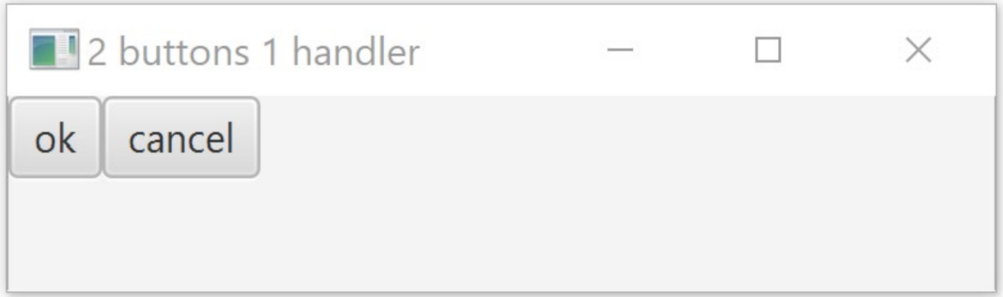
there is now a pane



two buttons with current class as event handler

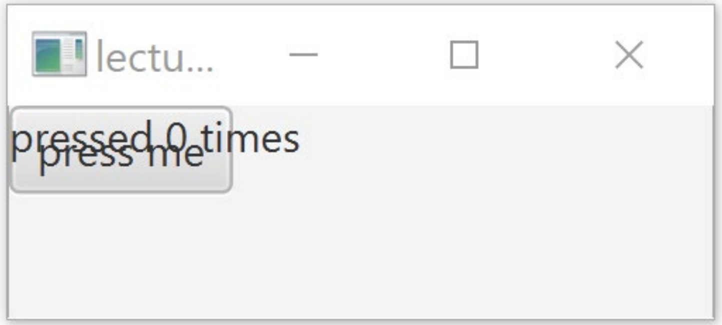
```
public class FXHandlerMain extends Application
                           implements EventHandler<ActionEvent> {
  public void start(Stage primaryStage) {
    Button btn1 = new Button("ok");
    btn1.setOnAction(this);
    Button btn2 = new Button("cancel");
    btn2.setOnAction(this);
    Scene scene = new Scene(new HBox (btn1, btn2), 200, 60);
    primaryStage.setTitle("2 buttons 1 handler");
    primaryStage.setScene(scene);
    primaryStage.show();
  }

  public void handle(ActionEvent event) {
    Button btn = (Button) event.getSource();
    System.out.println(btn.getText() + " pressed");
  }
}
```

we use an HBox to place the buttons next to each other



GUI layout

Need for a managing your layout



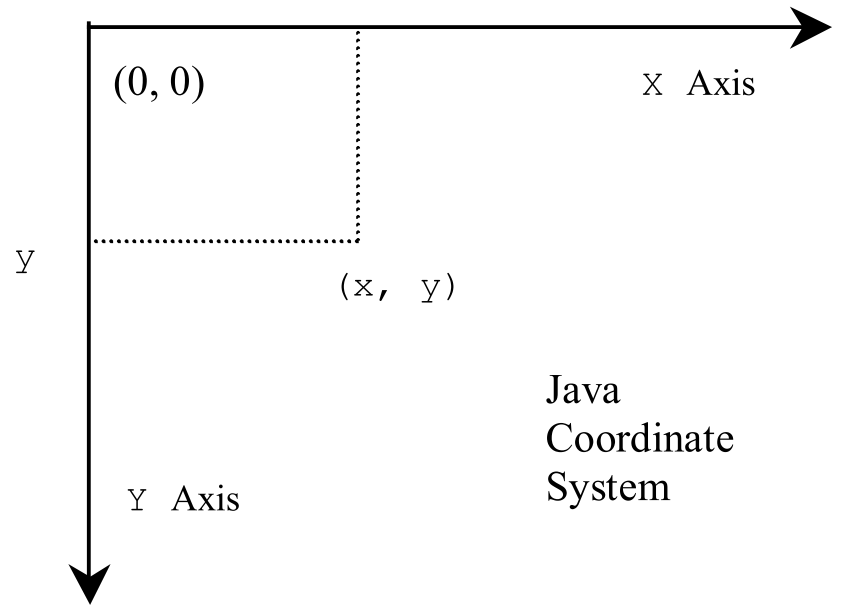
```
public class FXNoLayoutMain extends Application {
  IntegerProperty counter = new SimpleIntegerProperty(0);

  @Override
  public void start(Stage stage) {
    Label lbl = new Label();
    lbl.textProperty().bind(Bindings.concat("pressed ", counter, " times"));
    Button btn = new Button("press me");
    btn.setOnAction(e -> counter.set(counter.intValue() + 1));
    Pane root = new Pane();
    root.getChildren().addAll(btn, lbl);
    stage.setTitle(this.getClass().getName());
    stage.setScene(new Scene(root, 300, 250));
    stage.show();
  }
}
```

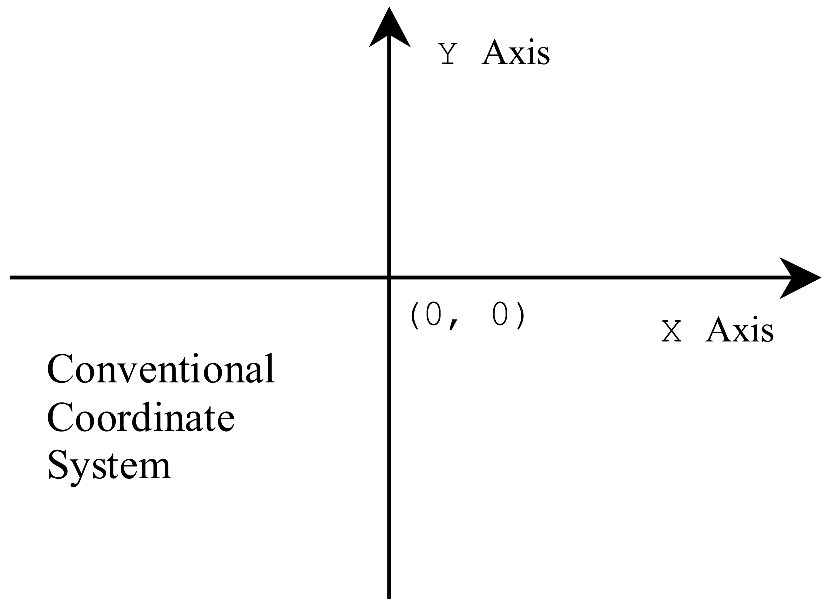
automatically update the label text when counter changes

handler changes the counter value

gui components are added to the pane

computer graphics scene coordinate system





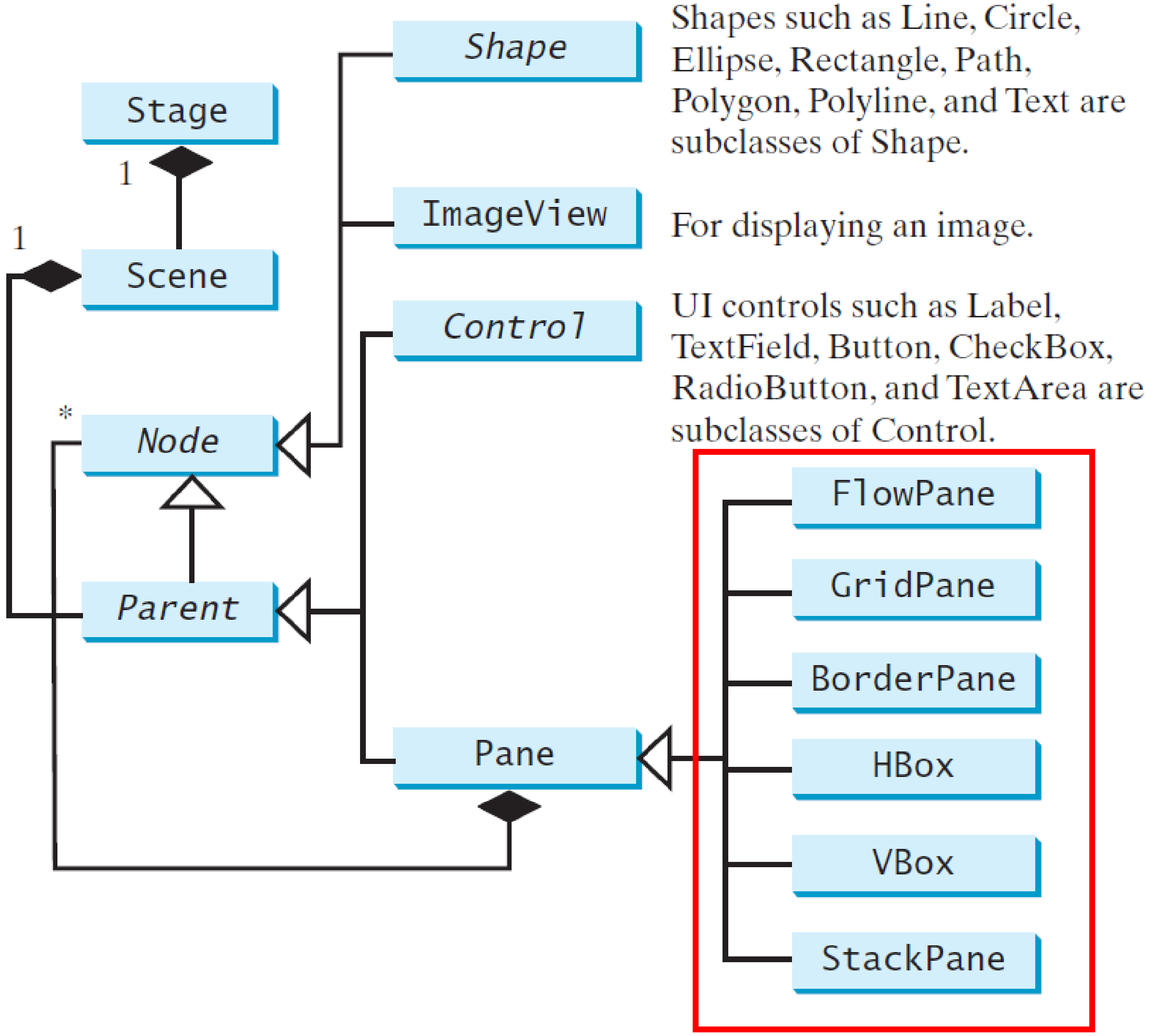
- Y-axis in the ‘wrong’ direction, origin in the top-left corner.
- This is counterintuitive to many people at first, and a source of mistakes!

layout in JavaFX

Different methods (can be combined):

1. let JavaFX compute position of Nodes
   - preferred way to handle simple layout
2. position Nodes using properties
   - compute layout (or size, …) based on properties of other Nodes
   - Java FX takes care of updating automatically
3. Do It Yourself
   - manipulate layout directly, used for fine-grained control
   - next lecture

Automatic scene layout using specialized panes


Shape
Shapes such as Line, Circle, Ellipse, Rectangle, Path, Polygon, Polyline, and Text are subclasses of Shape.
Stage
1
1
Scene
ImageView
For displaying an image.
Control
UI controls such as Label, TextField, Button, CheckBox, RadioButton, and TextArea are subclasses of Control.
*
Node
Parent
Pane
FlowPane
GridPane
BorderPane
HBox
VBox
StackPane

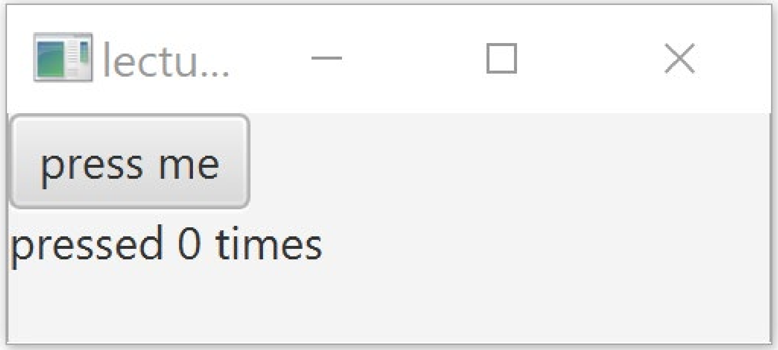
layout panes

name	description
Pane	base of Pane, no particular layout
StackPane	nodes in the center (on top of each other)
FlowPane	nodes next to each other, horizontally or vertically
HBox	single horizontal row
VBox	single vertical column
GridPane	matrix of cells to hold nodes
BorderPane	top, bottom, left, right, and centre region

`getChildren()` returns the (Observable!) list of nodes of the pane

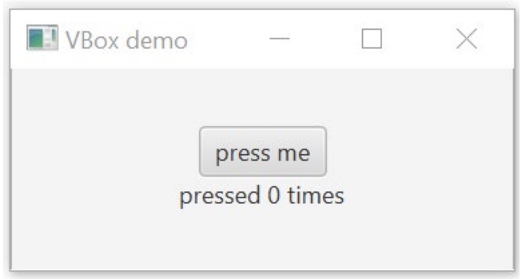
VBox for vertical layout

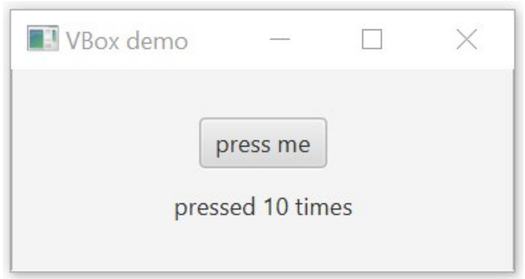
```
public class FXNoLayoutMain extends Application {
  IntegerProperty counter = new SimpleIntegerProperty(0);

  @Override
  public void start(Stage stage) {
    Label lbl = new Label();
    lbl.textProperty().bind(Bindings.concat("pressed ", counter, " times"));
    Button btn = new Button("press me");
    btn.setOnAction(e -> counter.set(counter.intValue() + 1));
    VBox root = new VBox();
    root.getChildren().addAll(btn, lbl);
    stage.setTitle(this.getClass().getName());
    stage.setScene(new Scene(root, 300, 250));
    stage.show();
  }
}
```

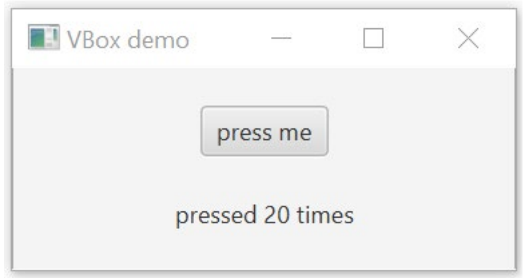


spacing & alignment options for VBox

```
public class FXNoLayoutMain extends Application {
  IntegerProperty counter = new SimpleIntegerProperty(0);

  @Override
  public void start(Stage stage) {
    Label lbl = new Label();
    lbl.textProperty().bind(Bindings.concat("pressed ", counter, " times"));
    Button btn = new Button("press me");
    VBox vbox = new VBox();
    vbox.getChildren().addAll(btn, lbl);
    vbox.setAlignment(Pos.CENTER);
    btn.setOnAction(e -> { counter.set(counter.intValue() + 1);
                           vbox.setSpacing(counter.doubleValue()); });
    root.getChildren().addAll(btn, lbl);
    stage.setTitle("VBox demo");
    stage.setScene(new Scene(root, 250, 100));
    stage.show();
  }
}
```



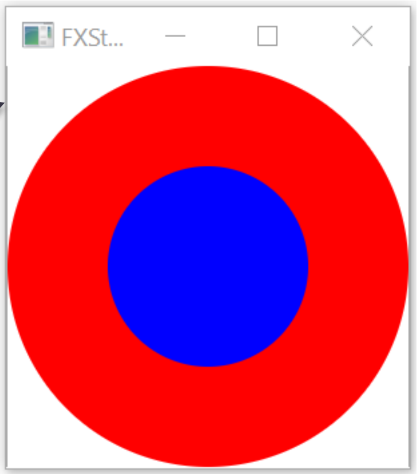




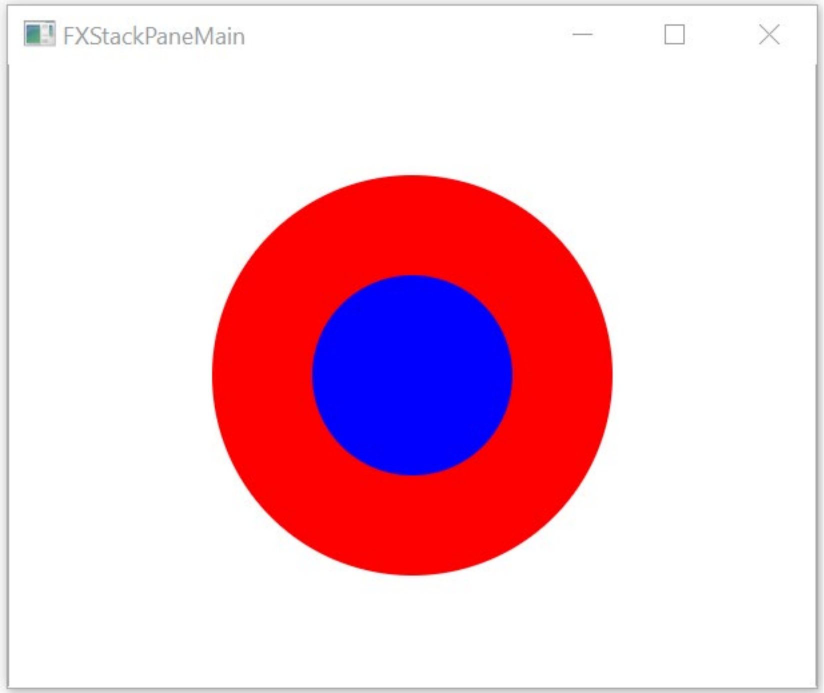
stack pane: everything centred and stacked

```
public void start(Stage stage) {
  Circle redCircle = new Circle(100);
  redCircle.setFill(Color.RED);
  Circle blueCircle = new Circle(50);
  blueCircle.setFill(Color.BLUE);
  Pane root = new StackPane(redCircle, blueCircle);
  stage.setTitle(this.getClass().getSimpleName());
  stage.setScene(new Scene(root));
  stage.show();
}
```

after startup



after resizing



nesting panes: border pane with stack panes

```
public void start(Stage stage) {
  Pane circles    = new StackPane();
  Pane rectangles = new StackPane();
  Pane root       = new BorderPane(null, circles, null, null, rectangles);
  Color[] colours = {Color.RED, Color.BLUE, Color.WHITE, Color.GREEN, Color.YELLOW};
  for (int i = colours.length; i > 0; i--) {
    circles.getChildren().add(new Circle(i * 20, colours[i - 1]));
    rectangles.getChildren().add(new Rectangle(i * 40, i * 20, colours[i - 1]));
  }
  stage.setTitle(this.getClass().getSimpleName());
  stage.setScene(new Scene(root));
  stage.show();
}
```

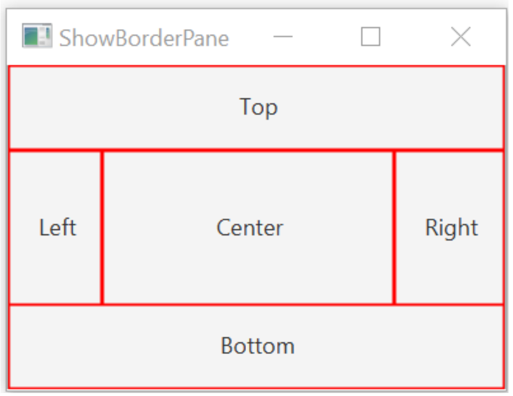
Center

Top

Right

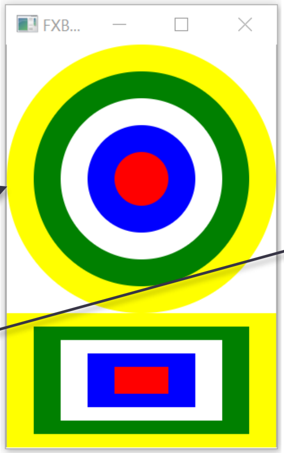
Bottom

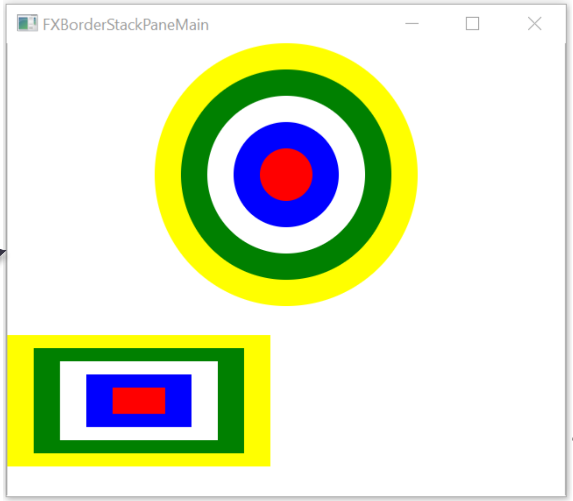
Left



after startup

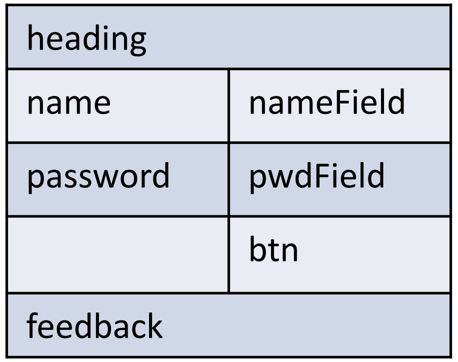
after resizing

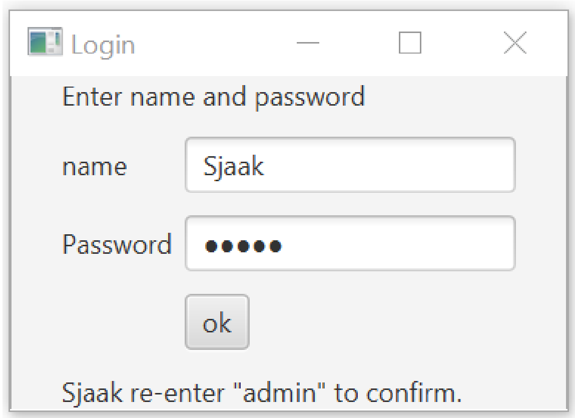




A possible login dialog using a grid pane

```
public class Login extends Application {
 private String pwd = "pwd";
 public void start(Stage stage) {
   GridPane grid = new GridPane();
   grid.setAlignment(Pos.CENTER);
   grid.setHgap(5);
   grid.setVgap(10);
   Label heading = new Label("Enter name and password");
   grid.add(heading, 0, 0, 2, 1); // spans 2 columns, 1 row.
   grid.add(new Label("name"), 0, 1);
   grid.add(new Label("Password"), 0, 2);
   TextField nameField = new TextField("user");
   TextField pwdField  = new PasswordField();
   grid.add(nameField, 1, 1);
   grid.add(pwdField, 1, 2);
   Label feedback = new Label();
   grid.add(feedback, 0, 4, 2, 1);
   Button btn = new Button();
   …
```





Enter name and password

name Sjaak

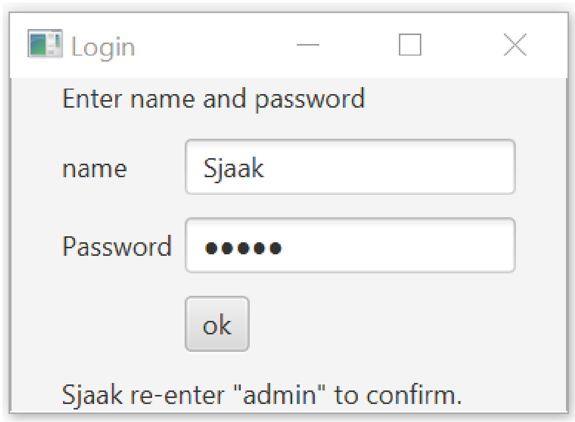
Password •••••

Sjaak re-enter "admin" to confirm.

A possible login dialog using a grid pane

```
public void start(Stage stage) {
    …
   btn.setText("ok");
   btn.setOnAction(e -> {
     String name       = nameField.getText();
     String pwdEntered = pwdField.getText();
     if (pwd.equals(pwdEntered)) {
       stage.close();
     } else {
       feedback.setText(name + " re-enter \"" + pwdEntered + "\" to confirm.");
       pwd = pwdEntered;
       pwdField.clear();
     }
   });
   grid.add(btn, 1, 3);
   Scene scene = new Scene(grid, 250, 150);
   stage.setTitle(this.getClass().getSimpleName());
   stage.setScene(scene);
   stage.show();
 }
```

heading	
name	nameField
password	pwdField
	btn
feedback	



Enter name and password

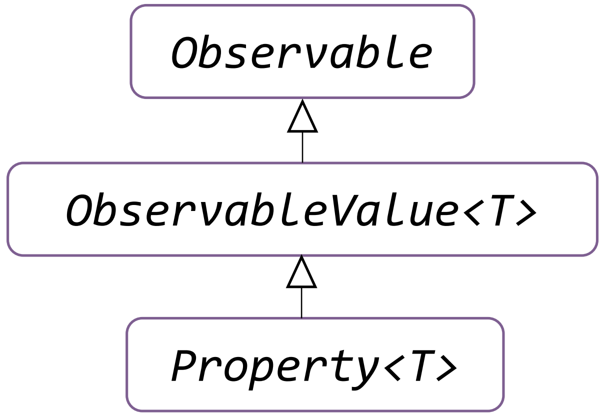
name Sjaak

Password ●●●●●

Sjaak re-enter "admin" to confirm.

Observables and listeners

Observables



- The signature of bind (as defined in interface Property<T>):

```
    void bind(ObservableValue<? extends T> observable)
```

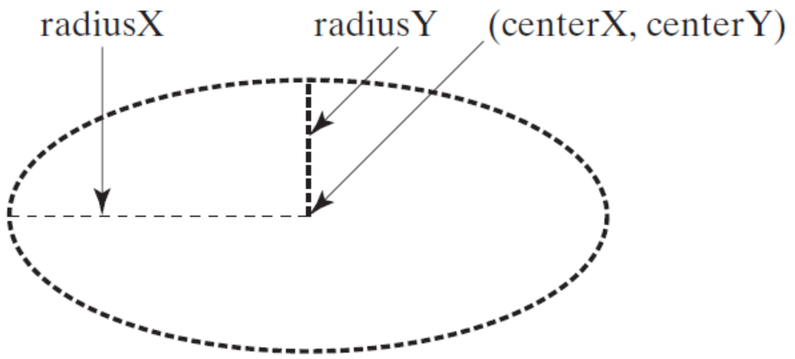
- An ObservableValue wraps a value and allows to observe the value for changes.
- When an ObservableValue changes it generates a change event:
  - a **change listener** is called (provided the change listener is installed)

```
interface ChangeListener<T> {
  void changed(ObservableValue<? extends T> observable, T oldValue, T newValue)
}
```

- To install a listener use the ObservableValue method

```
void addListener(ChangeListener<? super T> listener)
```

resize ellipse to fill Pane, using a listener



```
interface ChangeListener<T> {
  void changed(ObservableValue<? extends T> observable,
               T oldValue, T newValue)
}
```

Stackpane will keep ellipse in the center of the pane

```
public void start(Stage stage) {
  Ellipse ellipse = new Ellipse();
  ellipse.setFill(Color.RED);
  Pane root = new StackPane(ellipse);
  root.widthProperty().addListener((obs, ov, nv) -> ellipse.setRadiusX(nv.doubleValue()*0.45));
  root.heightProperty().addListener((obs, ov, nv) -> ellipse.setRadiusY(nv.doubleValue()*0.45));
  stage.setTitle(this.getClass().getSimpleName());
  stage.setScene(new Scene(root, 200, 100));
  stage.show();
}
```

Lecture 9: GUIs: JavaFX (II)