GUIs: JavaFX

Object-Oriented Programming Lecture 8

IJP (Liang): chapter 14-16, 31 (online material)

http://docs.oracle.com/javase/8/javafx/get-started-tutorial/index.html

April 06, 2021

Radboud University

IN·DEI·NOMINE·FELICITER

GRAPHICAL USER INTERFACES

why GUI-programming in this course

GUI = Graphical User Interface

1. it is important to know how to make a GUI
2. it uses a lot of the concepts introduced in this course

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 only use this

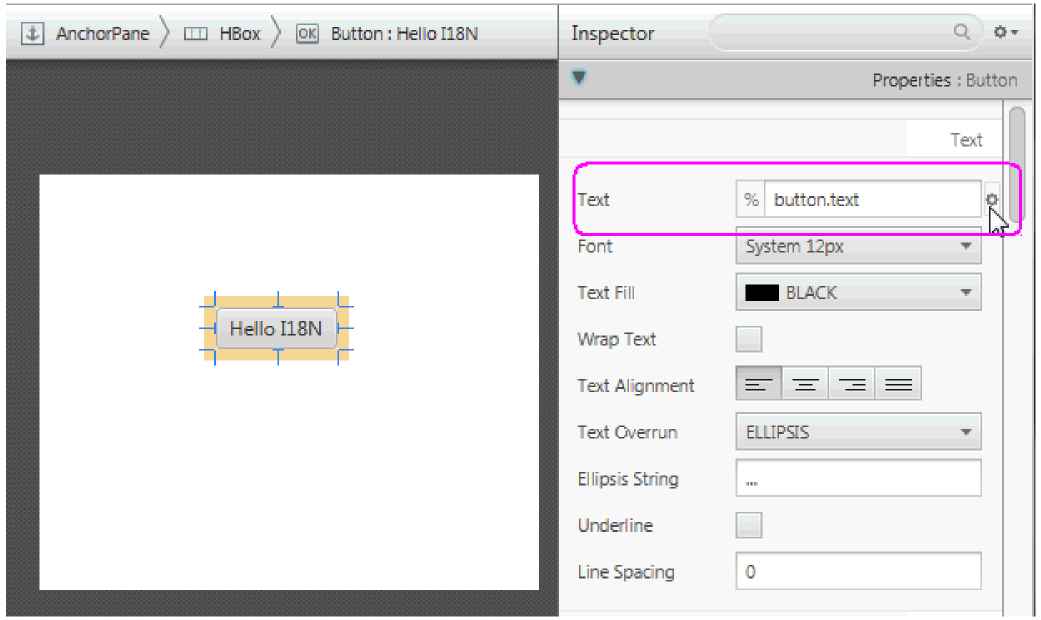
different ways of working with JavaFX

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 Java 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 (recursive tree) 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 APPLICATIONS

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

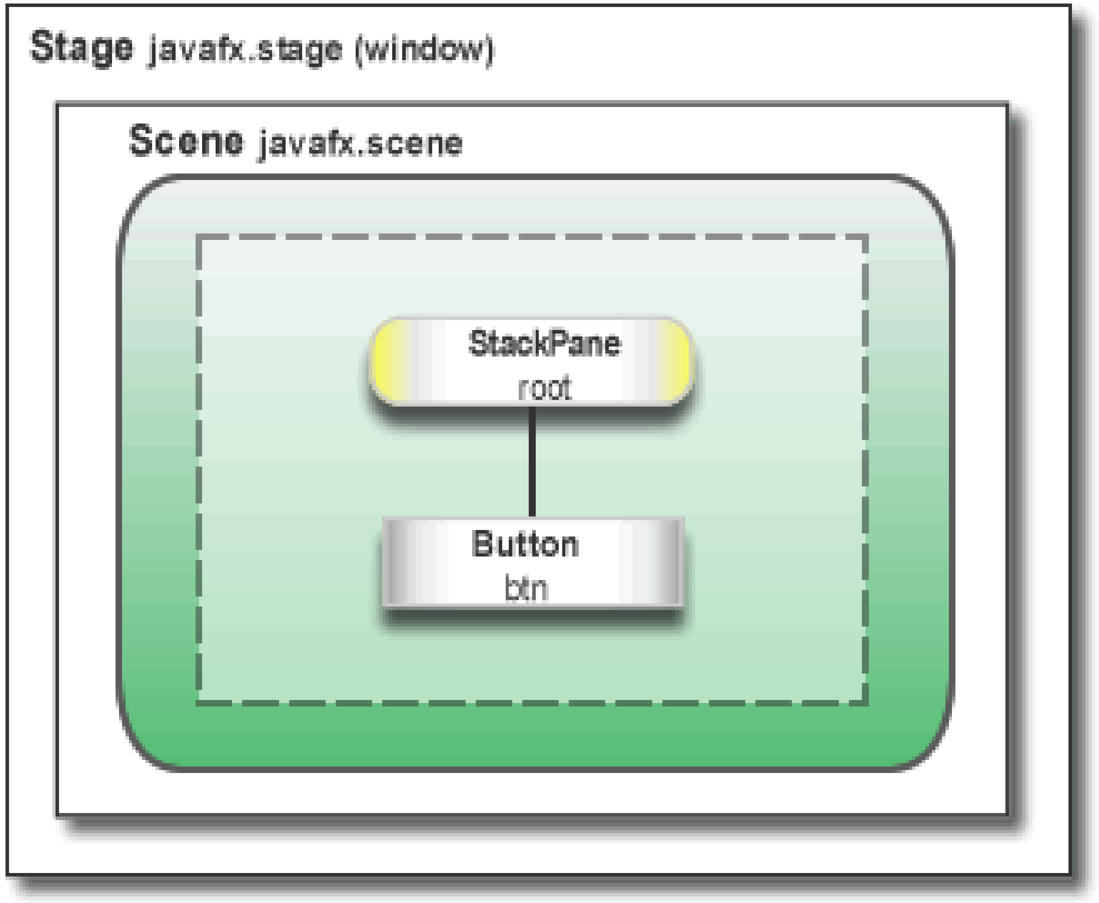
JavaFX `Application` life-cycle

what `launch()` does:

1. creates an instance of the specified Application class
2. calls the `init()` method
3. calls `start(javafx.stage.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

terminology

JavaFX has concepts that map to familiar GUI objects but have different names



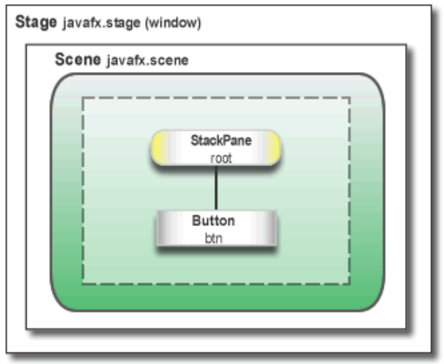
first JavaFX application

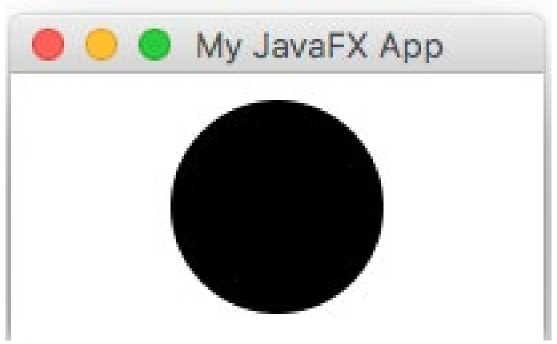
```
public class MyFirstJavaApp extends Application {
  public void start(Stage stage) {
    Circle circle = new Circle(100, 50, 40);
    Pane root = new Pane(circle);
    Scene scene = new Scene(root, 200, 100);
    stage.setTitle("My JavaFX App");
    stage.setScene(scene);
    stage.show();
  }
  public static void main(String[] args){
    launch(args);
  }
}
```

primary stage is always present

x, y, r

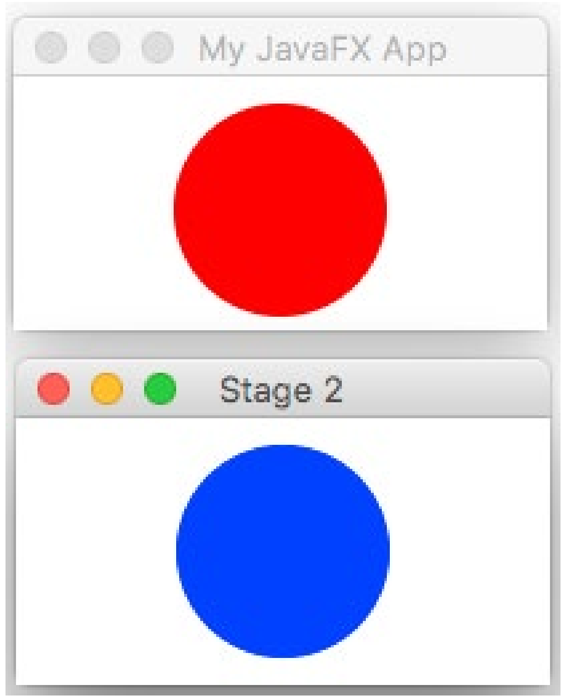
Width, height





two windows / Stages and Scenes

```
public void start(Stage stage) {
  Circle circle = new Circle(100, 50, 40);
  circle.setFill(Color.RED);
  Pane pane = new Pane(circle);
  Scene scene = new Scene(pane, 200, 100);
  stage.setTitle("My JavaFX App");
  stage.setScene(scene);
  stage.show();

  Stage stage2 = new Stage();
  stage2.setTitle("Stage 2");
  Circle circle2 = new Circle(80, 50, 40);
  circle2.setFill(Color.BLUE);
  stage2.setScene(new Scene(new StackPane(circle2), 200, 100));
  stage2.show();
}
```



JAVAFX ARCHITECTURE

Stage, Scene, Pane, Node

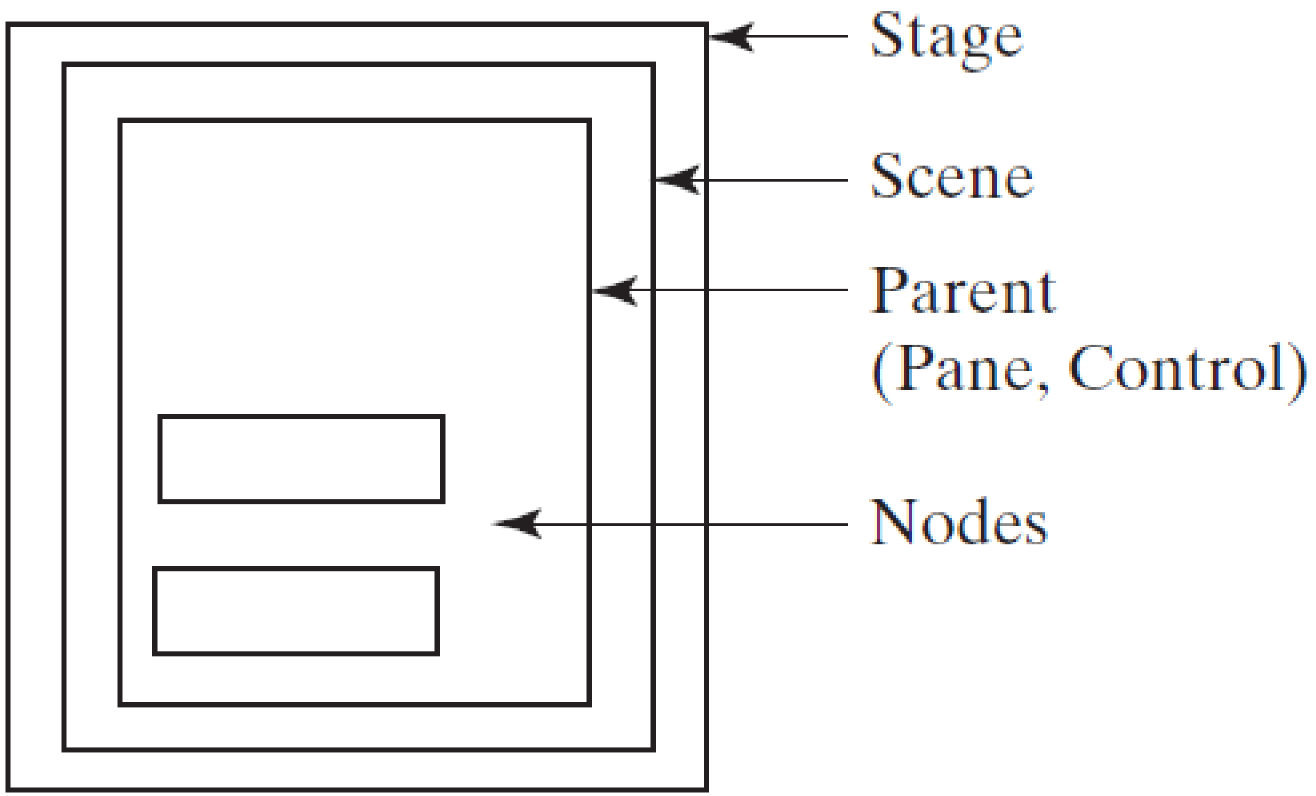
an application can have multiple `Stages`

`Stage` has one `Scene`

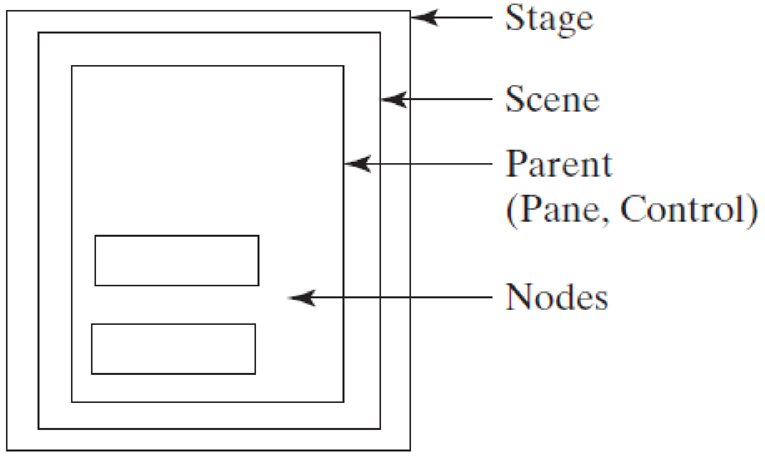
`Scene` has one `Parent` (root)

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

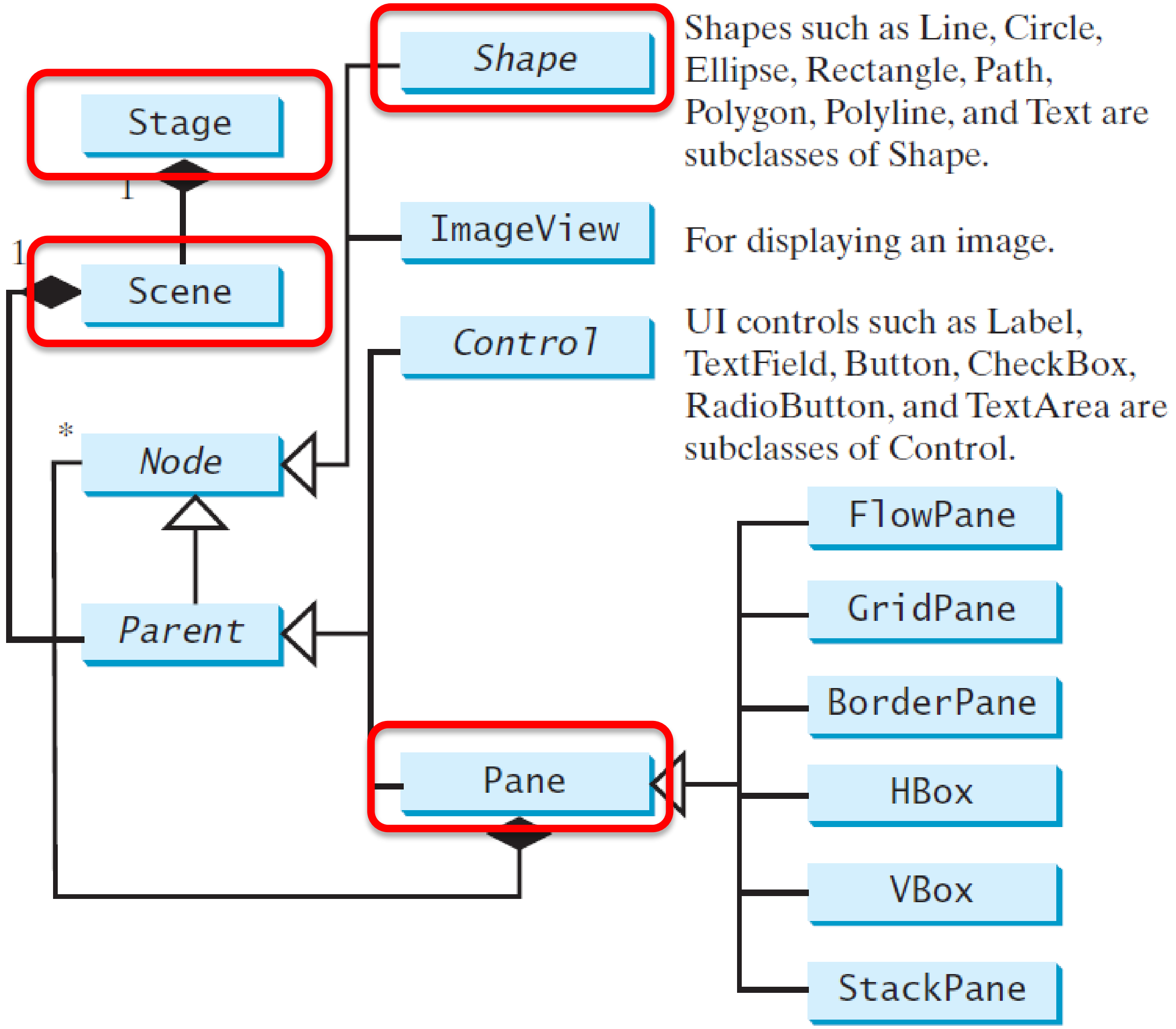
`Node`: any JavaFX component



scene, stage, nodes, ..



(a)

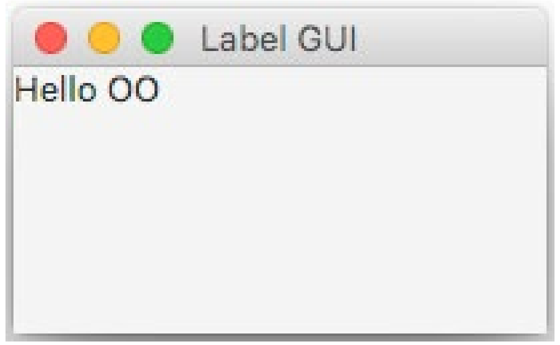


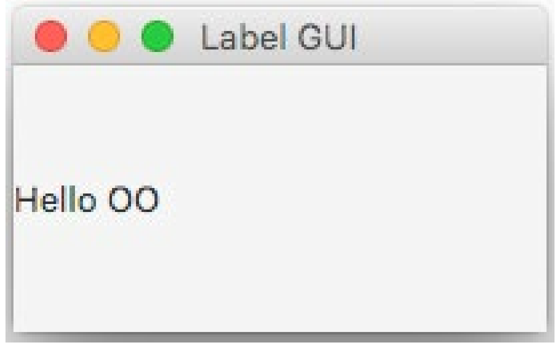
Stage
1
1
Scene
*
Node
Parent
Shape
ImageView
Control
Pane
FlowPane
GridPane
BorderPane
HBox
VBox
StackPane
Shapes such as Line, Circle, Ellipse, Rectangle, Path, Polygon, Polyline, and Text are subclasses of Shape.
For displaying an image.
UI controls such as Label, TextField, Button, CheckBox, RadioButton,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

(b)

displaying text / Label

```
public void start(Stage stage) {
  Label label = new Label("Hello OO");
  Pane pane = new Pane(label);
  stage.setTitle("Label GUI");
  stage.setScene(new Scene(pane, 200, 100));
  stage.show();
}
public void start(Stage stage) { // no Pane
  Label label = new Label("Hello OO");
  stage.setTitle("Label GUI");
  stage.setScene(new Scene(label, 200, 100));
  stage.show();
}
```





why do we ‘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 serious program has one or more `Pane` objects

- control layout
- set background color
- mouse handlers
- …

it is a good habit to always include a `Pane`

there are various `Pane` subclasses yielding different layout of nodes

MAKING THE GUI DO STUFF: PROPERTIES & EVENT HANDLING

properties

properties are (like) attributes

properties are Java objects containing/wrapping a value

properties are used instead of concrete types: e.g. `IntegerProperty` iso `int`

we can bind properties to other properties

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

when the source is changed all targets will be updated automatically

- cf the **observer pattern**

many attributes of JavaFX objects are properties

can be used to automatically update other objects when the property changes

properties: getters & setters

Objects with properties 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, e.g. `circle.centerXProperty()`
- **no** setter for the Property object – properties are mutated, not replaced
    - can be made **`final`**

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

```java
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());
}
```

Changes propagate down this chain!

Output

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

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

bidirectional binding demo: doubles

```
public static void run1() {
    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);
 }
```

Output

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

property demo: strings

Building observable Strings with embedded observables

```
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);
  print(s);
  y.bind(x.multiply(8).add(2));
  print(s);
  x.set(5);
  print(s);
}
private void static void print( StringProperty s ) {
      System.out.println(s.getValue());
}
```

Output

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

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

MVC (I)

Model + View tightly coupled

```
public class Model {
   private int modelAttr;
   private View viewAttr;

   public Model(int modAttr, View view) {
      this.modelAttr = modAttr;
      this.viewAttr  = view;
      viewAttr.setText("model attribute: " + modelAttr);
   }
   public int getModAttr() { return modelAttr; }
   public void setModAttr(int modAttr) {
      this.modelAttr = modAttr;
      viewAttr.setText("model attribute: " + modelAttr);
   }
}
```

```
public class View {
   Label label;

   public View(Label label) {
      this.label = label;
   }

   void setText (String txt ) {
      label.setText(txt);
   }
}
```

MVC (II)

Model + View disentangled using the observer pattern

```java
public class Model {
  private int modelAttr;
  private Observer<Model> viewAttr;

  public Model(int modAttr, View view) {
    this.modelAttr = modAttr;
    this.viewAttr  = view;
    viewAttr.update( this );
  }
  public int getModAttr() { return modelAttr; }
  public void setModAttr(int modAttr) {
    this.modelAttr = modAttr;
    viewAttr.update( this );
   }
}
```

```java
public interface Observer<T> {
   void update( T observable );
}
```

```java
public class View implements Observer<Model>{
  Label label;
  public View(Label label) {
    this.label = label;
  }
  void setText(String txt){ label.setText(txt);}

  @Override
  public void update(Model observable) {
    label.setText("Model: " + observable.getModAttr());
  }
}
```

MVC (III)

Model + View disentangled using properties

```java
public class Model {
  private final IntegerProperty modelAttr;
  public Model(int modAttr) {
    this.modelAttr = new SimpleIntegerProperty(modAttr);
  }

  public IntegerProperty modelAttrProperty () {
    return modelAttr;
  }

  public int getModelAttr() {
    return modelAttr.getValue();
  }

  public void setModelAttr(int modAttr) {
    modelAttr.setValue(modAttr);
  }}
```

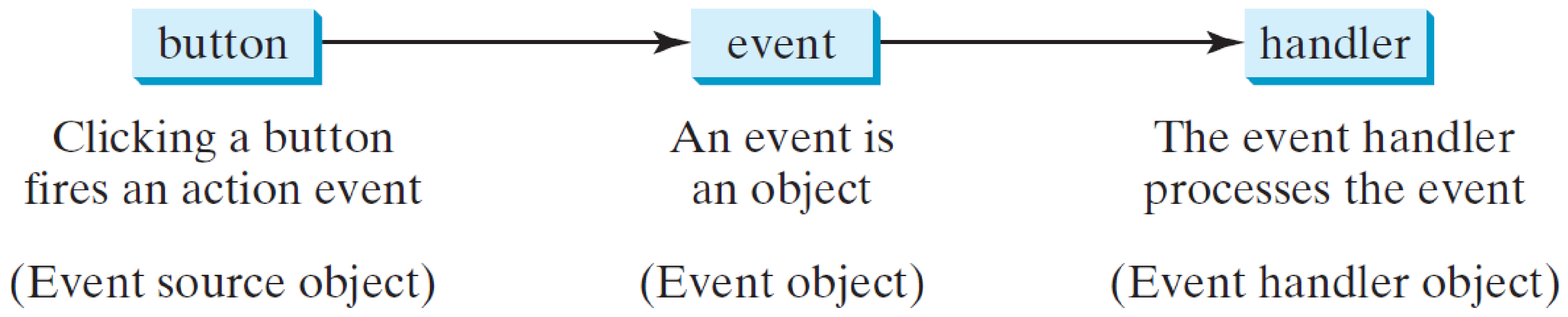
```java
public class View {
  private final Label label;
  public View(Label label) {
    this.label = label;
  }
  public StringProperty labelProperty(){
    return label.textProperty();
  }
}
```

MVC (IV)

Model + View tying things together

```
public class Main extends Application {
   @Override
   public void start(Stage stage) {
      Label label = new Label();
      View view = new View(label);
      Model model = new Model(42);
      view.labelProperty().bind(Bindings.concat("Model: ", model.modelAttrProperty()));
      Pane root = new StackPane(label);
      Scene scene = new Scene(root, 200, 100);
      stage.setTitle("MVC App");
      stage.setScene(scene);
      stage.show();
  }
}
```

handling (button) events



JavaFX takes care of generating the event object and passing it to an appropriate handler

we must specify the ***handler***

Functional Interface!
(Single Abstract Method)

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

handler always gets the `event` causing the call as its argument

implementing handlers

making specialized button subclasses with the desired functionality is inconvenient;
the **strategy pattern** with a handler strategy is more practical

handlers are the controllers in the MVC pattern

several ways to implement handlers, but always implementing `EventHandler`

1. an separate class implementing the **`interface`** `EventHandler`
2. the **`this`** object if it implements `EventHandler`
3. named inner-class implementing `EventHandler`
4. an anonymous class implementing `EventHandler`
5. lambda-expression for `EventHandler`

choice depends on size of handler and things to be known

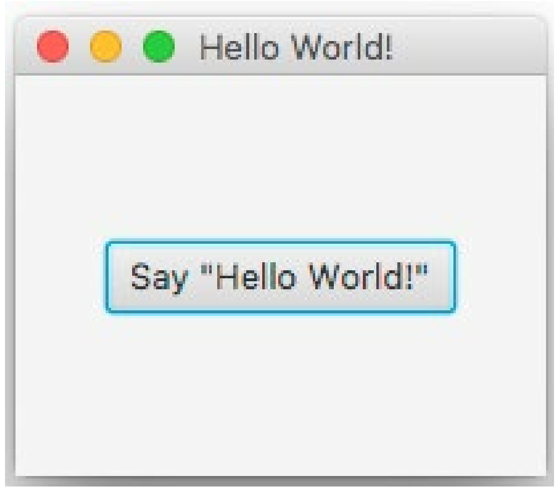
- for small things lambda-expressions and anonymous classes are very handy

a button with anonymous class as event handler

```
public void start(Stage stage) {
  Button btn = new Button();
  btn.setText("Say \"Hello World!\"");
  btn.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
      System.out.println("Hello World!");
    }
  });
  StackPane root = new StackPane();
  root.getChildren().add(btn);
  Scene scene = new Scene(root, 200, 150);
  stage.setTitle("Hello World!");
  stage.setScene(scene);
  stage.show();
}
```

anonymous class

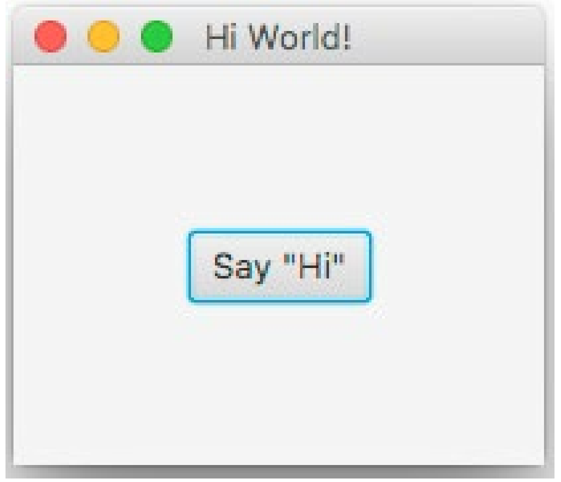
handler strategy



a button with lambda expression as event handler

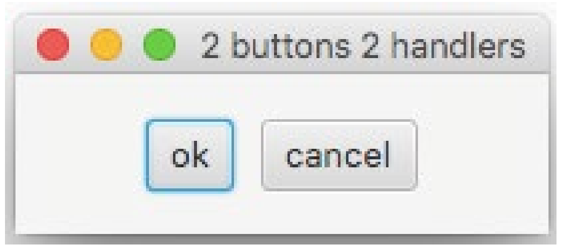
```java
public void start(Stage stage) {
  Button btn = new Button();
  btn.setText("Say \"Hi\"");
  btn.setOnAction(e -> System.out.println("Hi"));
  StackPane root = new StackPane(btn);
  Scene scene = new Scene(root, 200, 150);
  stage.setTitle("Hi World!");
  stage.setScene(scene);
  stage.show();
}
```

handler strategy as lambda expression



2 buttons 2 handlers

```
public void start(Stage primaryStage) {
  Button btn1 = new Button("ok");
  btn1.setOnAction(new OkHandler());
  Button btn2 = new Button("cancel");
  btn2.setOnAction(new CancelHandler());
  HBox root = new HBox();
  root.setAlignment(Pos.CENTER);
  root.setSpacing(10);
  root.getChildren().addAll(btn1, btn2);
  primaryStage.setTitle("2 buttons 2 handlers");
  primaryStage.setScene(new Scene(root, 200, 60));
  primaryStage.show();
}
private class OkHandler implements EventHandler<ActionEvent> {
  public void handle(ActionEvent event) {
    System.out.println("ok pressed");
} }
private class CancelHandler implements EventHandler<ActionEvent> {
  public void handle(ActionEvent event) {
    System.out.println("cancel pressed");
} }
```

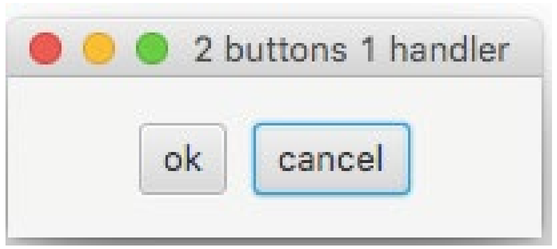


Output

```
ok pressed
cancel pressed
```

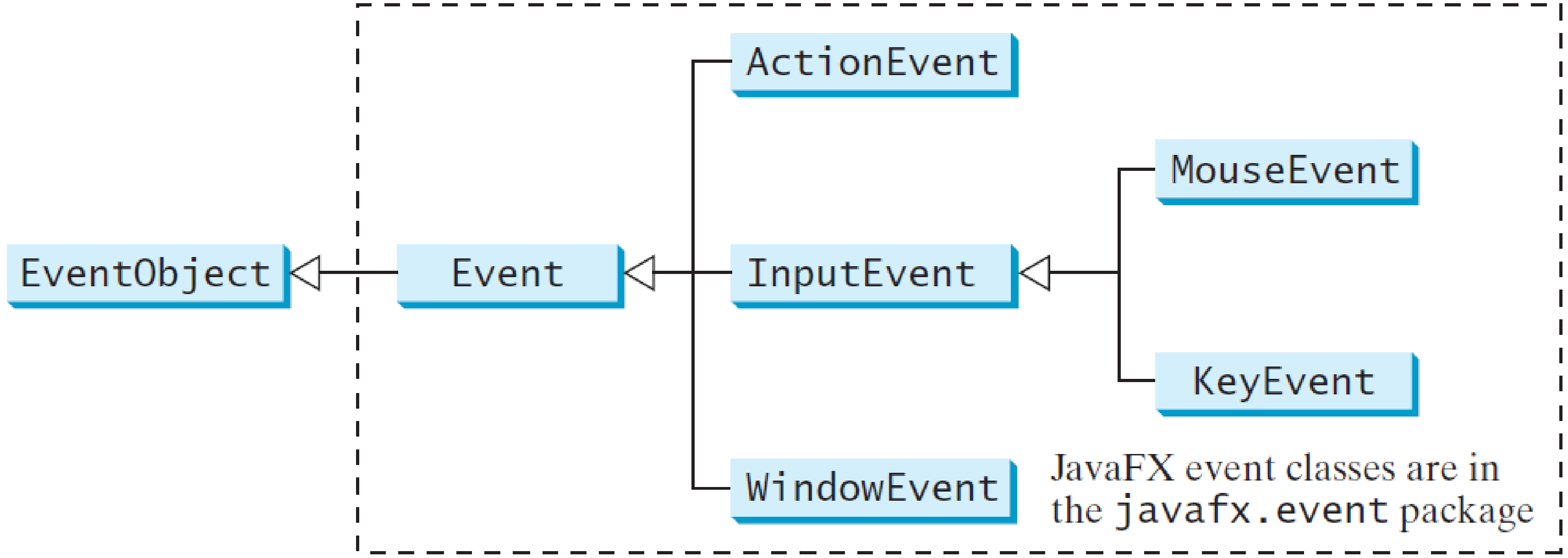
2 buttons 1 handler

```
public class FX2but1hndlr 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);
    HBox root = new HBox();
    root.setAlignment(Pos.CENTER);
    root.setSpacing(10);
    root.getChildren().addAll(btn1, btn2);
    Scene scene = new Scene(root, 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");
  }
```



Output
```
ok pressed
cancel pressed
```

event types



event objects contain specific information

- source
- position
- key
- …

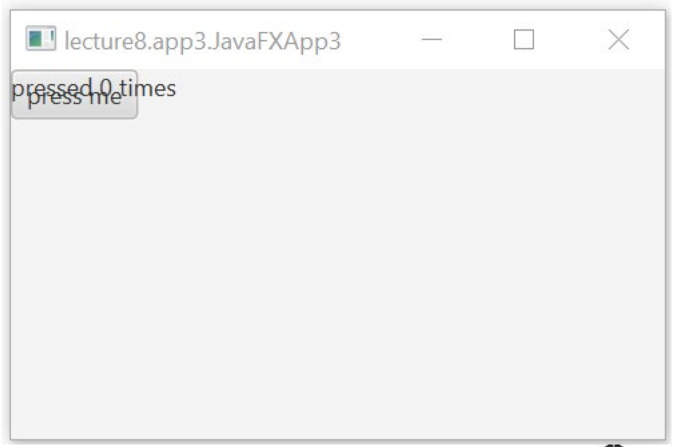
GUI LAYOUT

the need for a managing your layout

```
public class JavaFXApp3 extends Application {
  IntegerProperty counter = new SimpleIntegerProperty(); // should be in a model

  @Override
  public void start(Stage primaryStage) {
    Label  lbl = new Label("press the button");
    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);
    primaryStage.setTitle(this.getClass().getName());
    primaryStage.setScene(new Scene(root, 300, 250));
    primaryStage.show();
  }
}
```

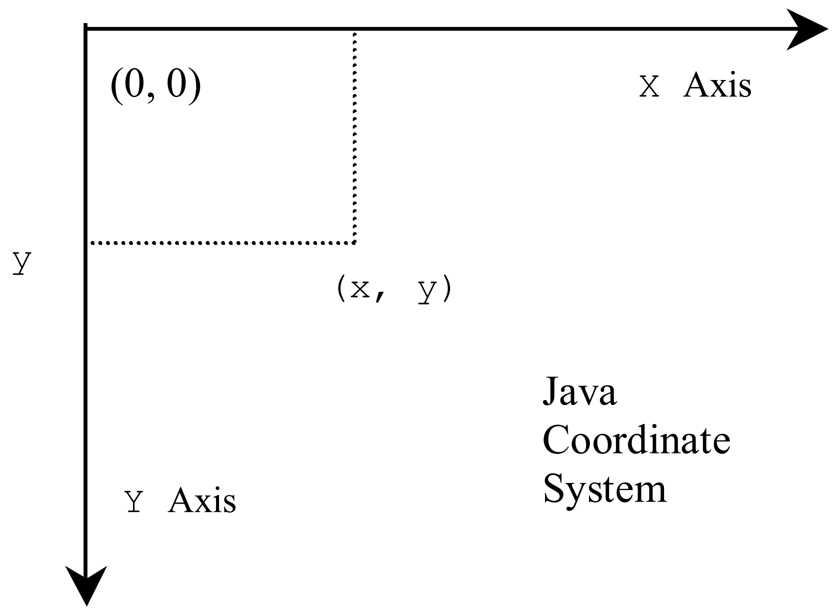
Could also use a **Text** (**Shape**)

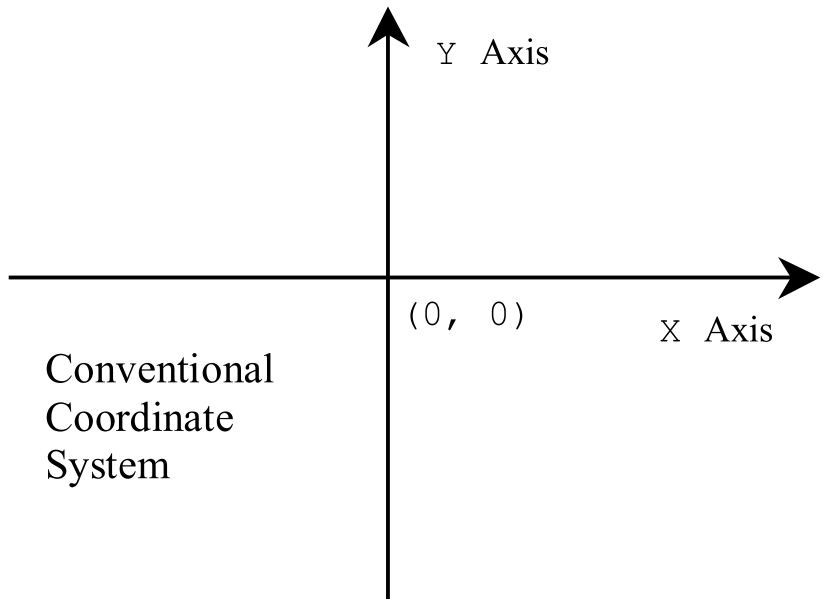
Updating the textProperty automatically updates the label



computer graphics scene coordinate system

Y-axis in the 'wrong' direction, origin in the top-left corner.





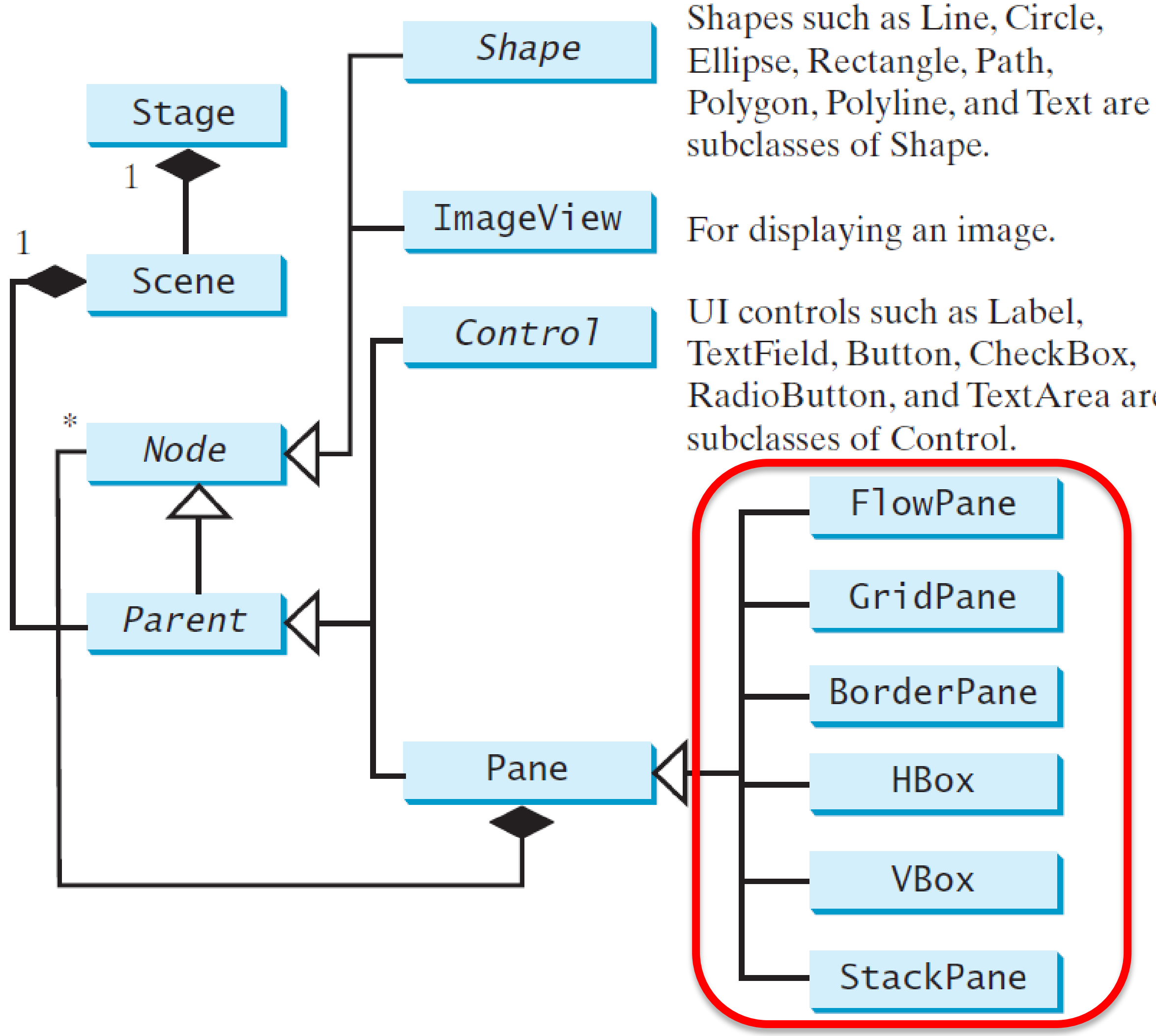
this is counterintuitive to many people at first, and a source of mistakes!

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 Node
   - 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 panes



Shapes such as Line, Circle, Ellipse, Rectangle, Path, 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.

layout Panes

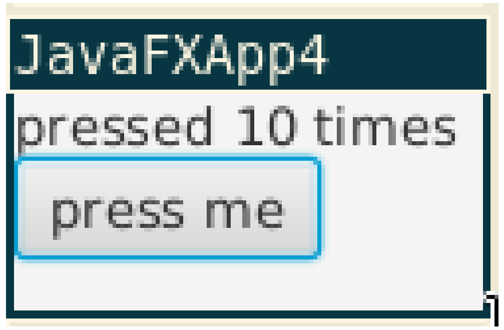
name	description
`Pane`	base of Pane, no particular layout
`StackPane`	nodes in the center (on top of each other)
`FlowPane`	nodes next to each other, horizontally or vertically
`HBox`	single horizontal row
`VBox`	single vertical column
`GridPane`	matrix of cells to hold nodes
`BorderPane`	top, bottom,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

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

VBox for vertical layout

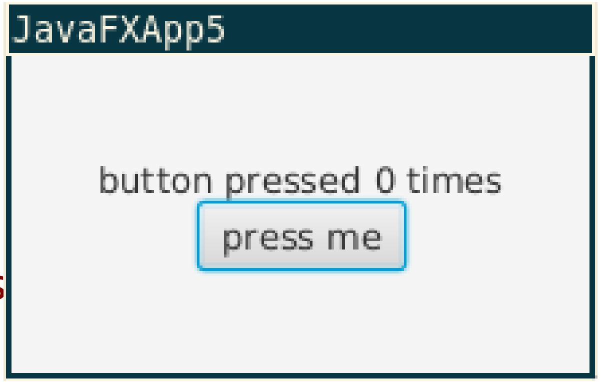
```
public class JavaFXApp4 extends Application {
  IntegerProperty counter = new SimpleIntegerProperty();

  @Override
  public void start(Stage primaryStage) {
    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 vbox = new VBox();
    vbox.getChildren().addAll(lbl, btn);
    Scene scene = new Scene(vbox, 150, 100);
    primaryStage.setTitle(this.getClass().getSimpleName());
    primaryStage.setScene(scene);
    primaryStage.show();
  }
}
```

layout pane

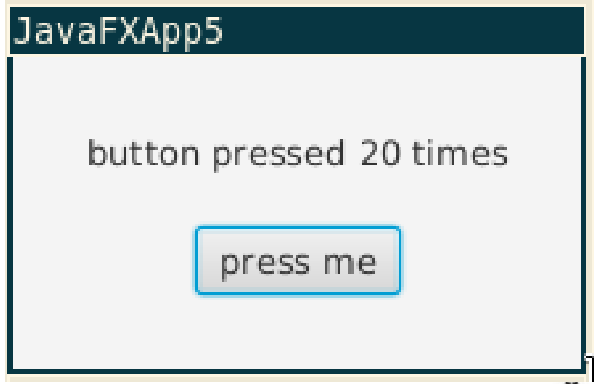


spacing & alignment options for VBox

```
public class JavaFXApp5 extends Application{
  private int counter = 0;
  @Override
  public void start(Stage primaryStage) {
    Label lbl  = new Label("button pressed " + counter + " times
    VBox vbox  = new VBox();
    Button btn = new Button("press me");
    btn.setOnAction(e -> {
      counter += 1;
      lbl.setText("button pressed " + counter + " times");
      vbox.setSpacing(counter);
    });
    vbox.getChildren().addAll(lbl, btn);
    vbox.setAlignment(Pos.CENTER);
    primaryStage.setTitle(this.getClass().getSimpleName());
    primaryStage.setScene(new Scene(vbox, 200, 150));
    primaryStage.show();
  }
}
```

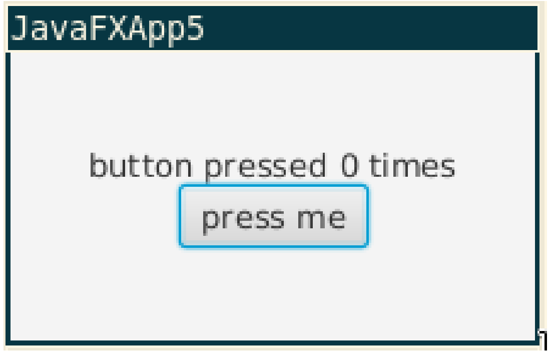


Without properties, we're doing all updating of both model & view in the handler

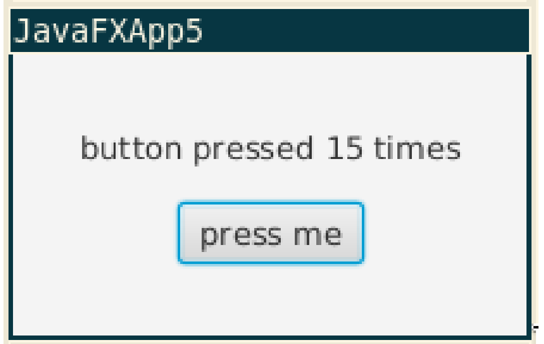


VBox spacing with a property

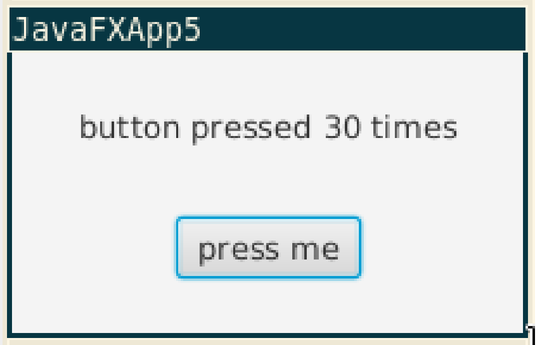
```java
public class JavaFXApp5 extends Application{
  private IntegerProperty counter = new SimpleIntegerProperty();

  @Override
  public void start(Stage stage) {
    Label lbl = new Label();
    lbl.textProperty().bind(Bindings.concat("button pressed ", counter, " times"));
    VBox vbox = new VBox();
    vbox.spacingProperty().bind(counter);
    Button btn = new Button("press me");
    btn.setOnAction(e -> counter.set(counter.intValue() + 1) );
    vbox.getChildren().addAll(lbl, btn);
    vbox.setAlignment(Pos.CENTER);
    stage.setTitle(this.getClass().getSimpleName());
    stage.setScene(new Scene(vbox, 200, 150););
    stage.show();
  }
}
```


JavaFXApp5

button pressed 0 times


JavaFXApp5

button pressed 15 times


JavaFXApp5

button pressed 30 times

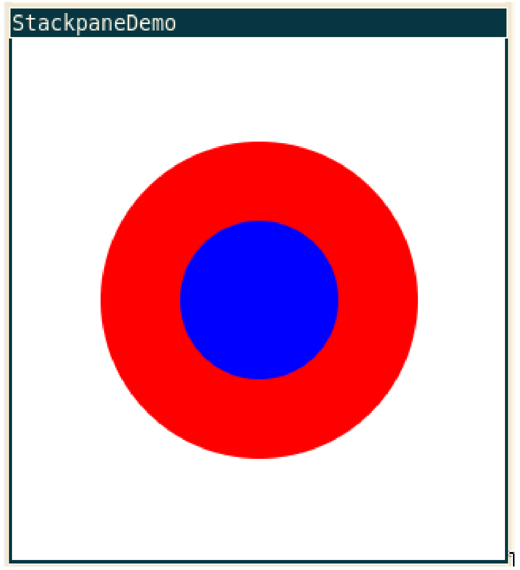
VBox spacing & label text automatically changed

stack pane: everything centred and stacked

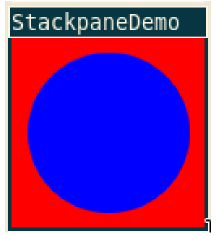
```
public void start(Stage stage) {
  Circle redCircle = new Circle(100);
  redCircle.setFill(Color.RED);
  Circle blueCircle = new Circle(50);
  blueCircle.setFill(Color.BLUE);
  Pane root = new StackPane(redCircle, blueCircle);
  stage.setTitle(this.getClass().getSimpleName());
  stage.setScene(new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

radius

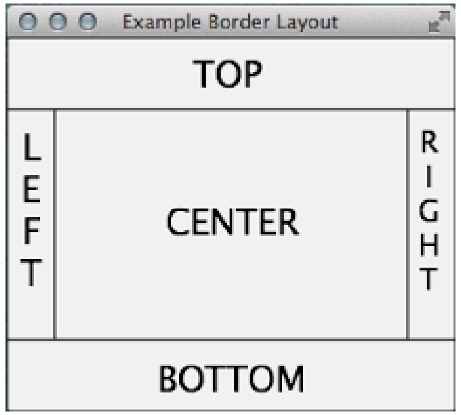
everything centred



centred after resize



nesting panes



Centre

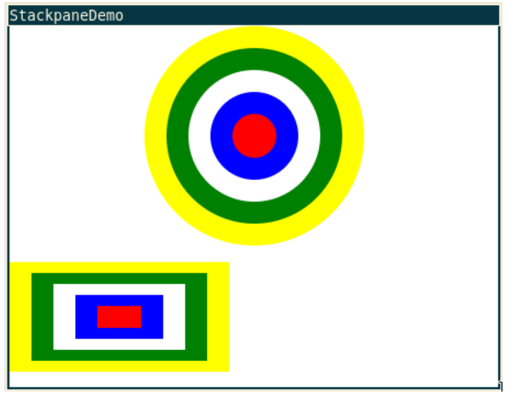
Top

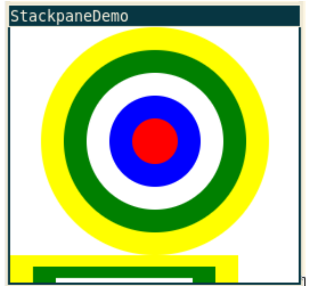
Right

Bottom

Left

```
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 = 5; 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();
}
```

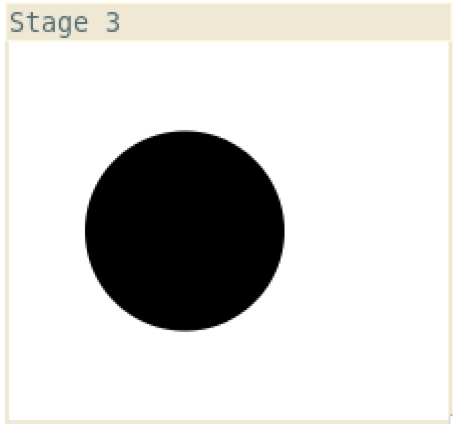


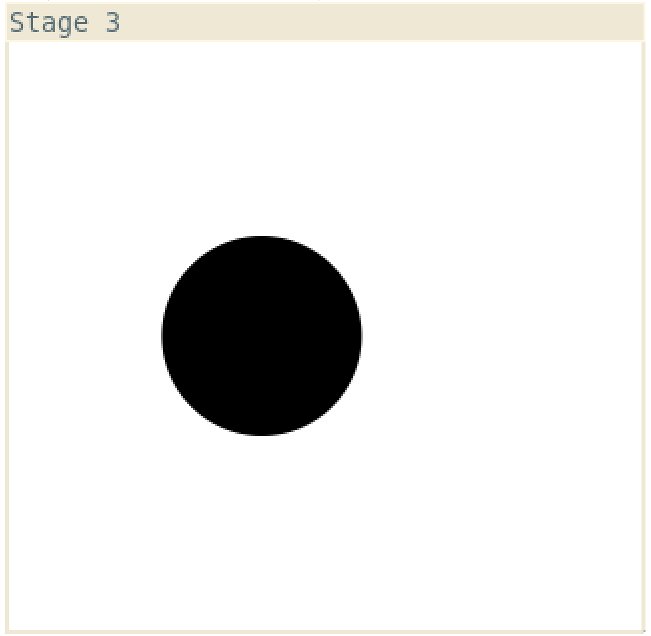


centring a circle with properties

```
public void start(Stage stage) {
  stage.setTitle("Stage 3");
  Circle circle3 = new Circle(50);
  circle3.centerXProperty().bind(stage.widthProperty().multiply(0.5));
  circle3.centerYProperty().bind(stage.heightProperty().multiply(0.5));
  stage.setScene(new Scene(new Pane(circle3), 200, 200));
  stage.show();
}
```

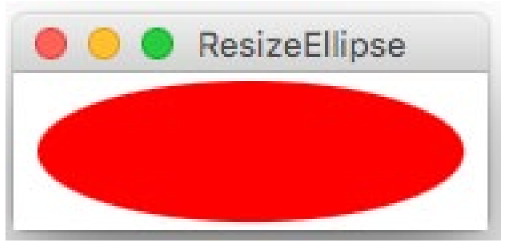
from Stage





GUI LAYOUT:
PERFORMING ACTIONS ON RESIZE

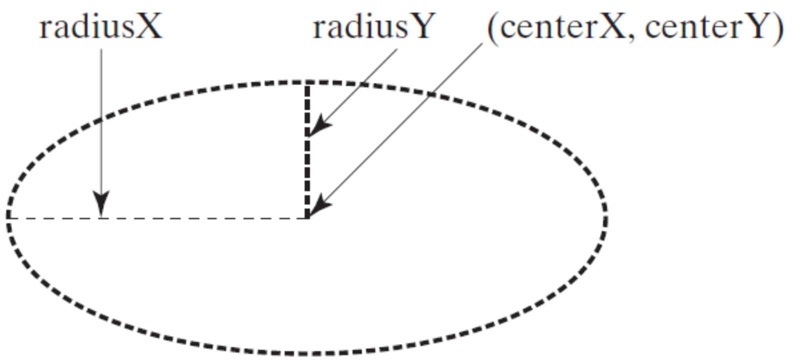
resize ellipse to fill StackPane

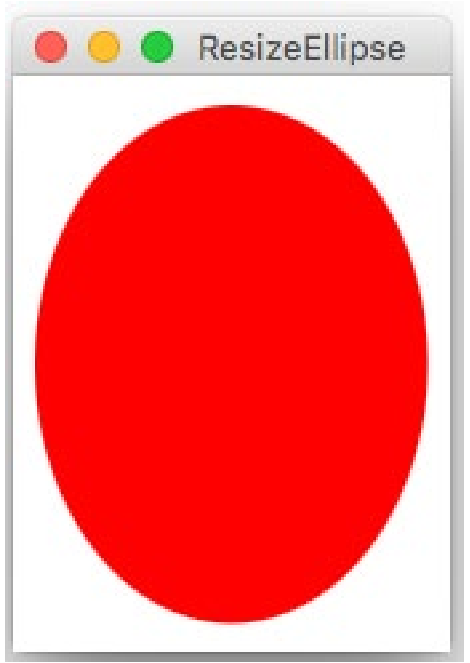


standard approach:

1. align centre with StackPane
2. handlers for change of height and width properties
   - determine current size of pane
   - adjust radius of ellipse

binding to properties can replace the handler





ad-hoc property “bindings” / handlers

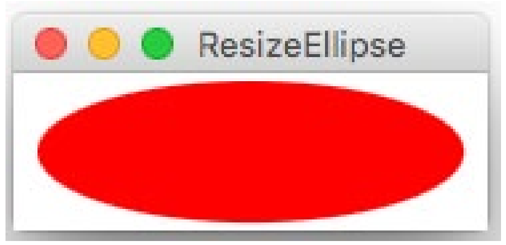
you can use properties even if there is no plain binding available

a **change listener** is called whenever the property changes

e.g.: a text-field that only allows integer values:

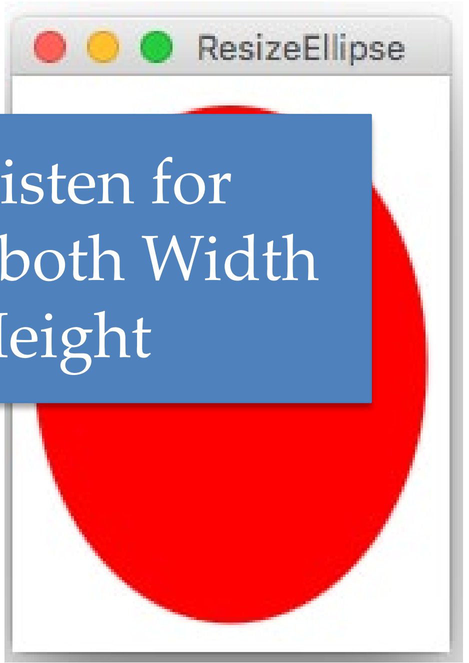
```
TextField text = new TextField(v.getValue().toString());
text.textProperty().addListener(new ChangeListener<String>() {
  @Override
  public void changed(ObservableValue<? extends String> observable,
                      String oldValue, String newValue) {
    if (! newValue.matches("-?\\d{1,8}")) {
      text.setText(oldValue);
    }
});
```

resize ellipse to fill StackPane: using a handler



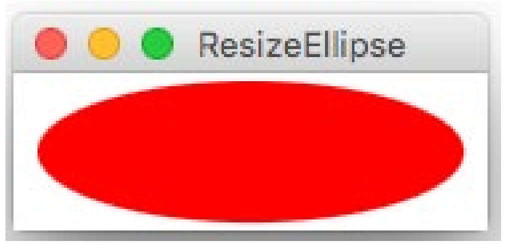
```
public class ResizeEllipse extends Application {
  private Ellipse ellipse;
  private Pane root;

  public void start(Stage stage) {
    ellipse = new Ellipse();
    ellipse.setFill(Color.RED);
    root = new StackPane(ellipse);
    ChangeListener<Number> onResize = new ResizeHandler();
    root.widthProperty().addListener(onResize);
    root.heightProperty().addListener(onResize);
    stage.setTitle(this.getClass().getSimpleName());
    stage.setScene(new Scene(root, 200, 100));
    stage.show();
  }
```

Create an instance of a `ChangeListener`



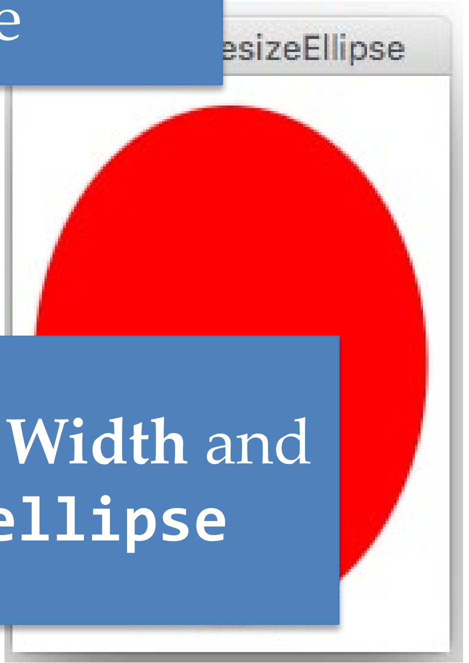
Make it listen for changes on both Width and Height

resize ellipse to fill StackPane: using a handler



```
public class ResizeEllipse extends Application {
  private Ellipse ellipse;
  private Pane root;
    ...
    ResizeHandler onResize = new ResizeHandler();
    root.widthProperty().addListener(onResize);
    root.heightProperty().addListener(onResize);
    ...

  private class ResizeHandler<T> implements ChangeListener<T> {
    @Override
    public void changed(ObservableValue<? extends T> ov, T t, T t1) {
      ellipse.setRadiusX(root.getWidth() * 0.45);
      ellipse.setRadiusY(root.getHeight() * 0.45);
    }
  }
}
```

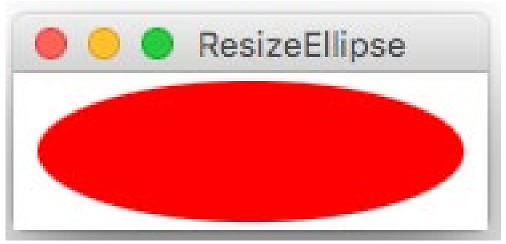
Gets called every time the **Width** or **Height** of **root** change



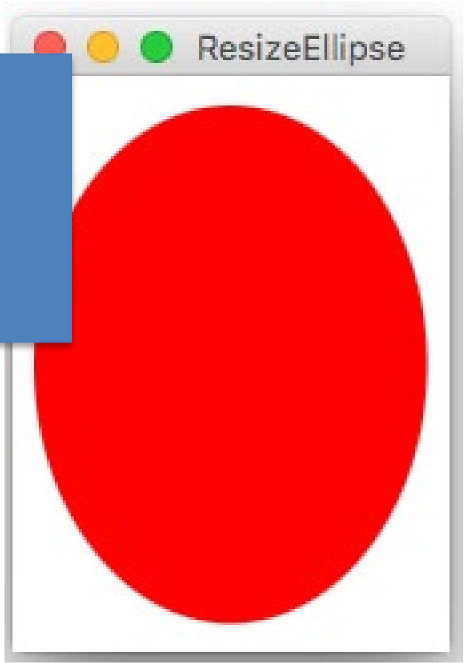
Updates both **Width** and **Height** of **ellipse**

resize ellipse to fill StackPane: using separate handlers

```
public class ResizeEllipse extends Application {
  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.intValue() * 0.45));
    root.heightProperty().addListener((obs, ov, nv) -> ellipse.setRadiusY(nv.intValue() * 0.45));
    stage.setTitle(this.getClass().getSimpleName());
    stage.setScene(new Scene(root, 200, 100));
    stage.show();
  }
}
```

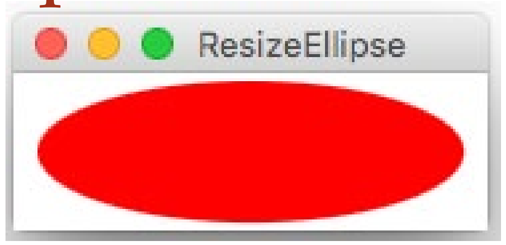


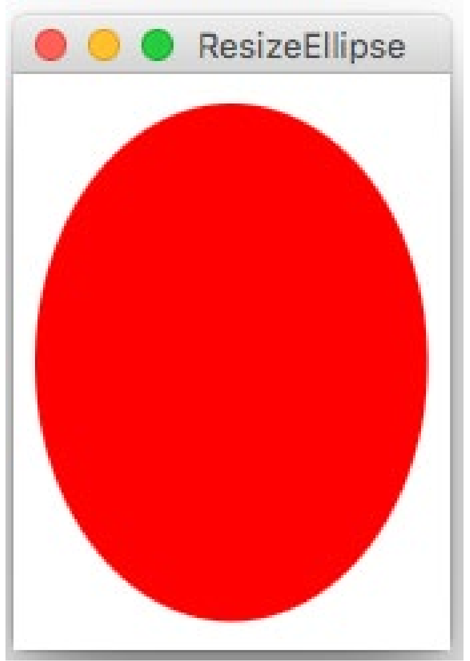
Only changes the relevant dimension



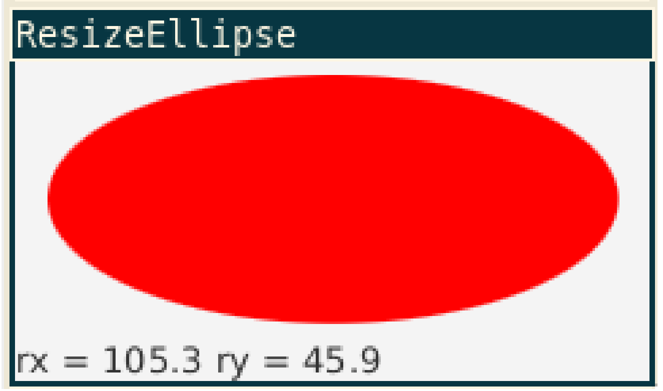
resize ellipse to fill StackPane: just bind the properties

```java
public class ResizeEllipse extends Application {
  public void start(Stage stage) {
    Ellipse ellipse = new Ellipse();
    ellipse.setFill(Color.RED);
    Pane root = new StackPane(ellipse);
    ellipse.radiusXProperty().bind(root.widthProperty().multiply(0.45));
    ellipse.radiusYProperty().bind(root.heightProperty().multiply(0.45));
    stage.setTitle(this.getClass().getSimpleName());
    stage.setScene(new Scene(root, 200, 100));
    stage.show();
  }
}
```

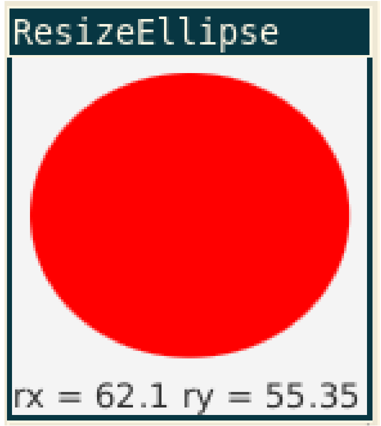




resize ellipse to fill StackPane: property changes propagate!



rx = 105.3 ry = 45.9



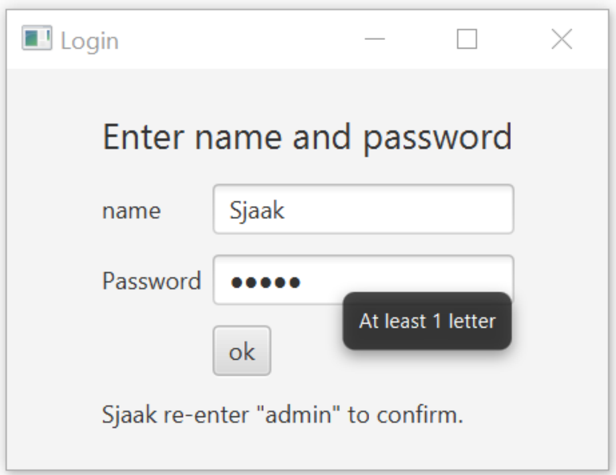
rx = 62.1 ry = 55.35

```
public class ResizeEllipse extends Application {
  public void start(Stage stage) {
    Ellipse ellipse = new Ellipse();
    ellipse.setFill(Color.RED);
    Pane elPane = new StackPane(ellipse);
    ellipse.radiusXProperty().bind(elPane.widthProperty().multiply(0.45));
    ellipse.radiusYProperty().bind(elPane.heightProperty().multiply(0.45));

    Label x = new Label(), y = new Label();
    x.textProperty().bind(Bindings.concat("rx = ", ellipse.radiusXProperty()));
    y.textProperty().bind(Bindings.concat("ry = ", ellipse.radiusYProperty()));
    Pane root = new BorderPane(elPane, null, null, new FlowPane(4, 4, x, y), null);
    stage.setTitle(this.getClass().getSimpleName());
    stage.setScene(new Scene(root, 200, 100));
    stage.show();
  }
}
```

DOING SOMETHING WITH A GUI:
A POSSIBLE LOGIN DIALOG

GridPane (used in the assignment)

```
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");
    heading.setFont(Font.font(18));
    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();
    pwdField.setTooltip(new Tooltip("At least 1 letter"));
    grid.add(nameField, 1, 1);
    grid.add(pwdField,  1, 2);
```

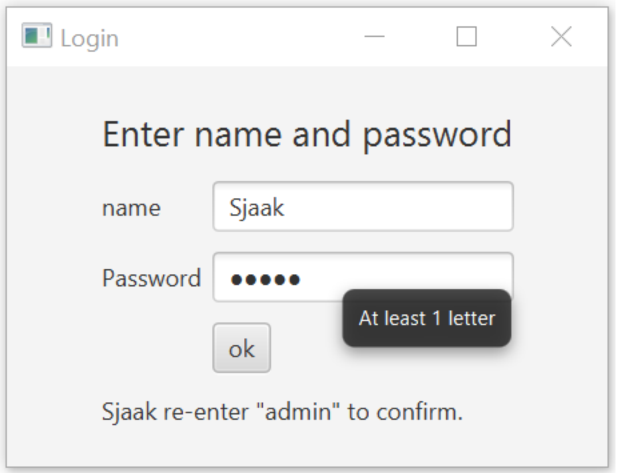


Sjaak re-enter "admin" to confirm.

heading	
name	nameField
password	pwdField
	btn
feedback	

GridPane

```
  Label feedback = new Label("");
  grid.add(feedback, 0, 4, 2, 1);
  Button btn = new Button();
  btn.setText("ok");
  btn.setOnAction(e -> {
    String name    = nameField.getText();
    String pwdUser = pwdField.getText();
    if (pwdUser.equals(pwd)) {
      stage.close();
    } else {
      feedback.setText(name + " re-enter \"" + pwdUser + "\" to confirm.");
      pwd = pwdUser;
      pwdField.clear();
    }
  });
  grid.add(btn, 1, 3);
  Scene scene = new Scene(grid, 300, 200);
  stage.setTitle(this.getClass().getSimpleName());
  stage.setScene(scene);
  stage.show();
}
```



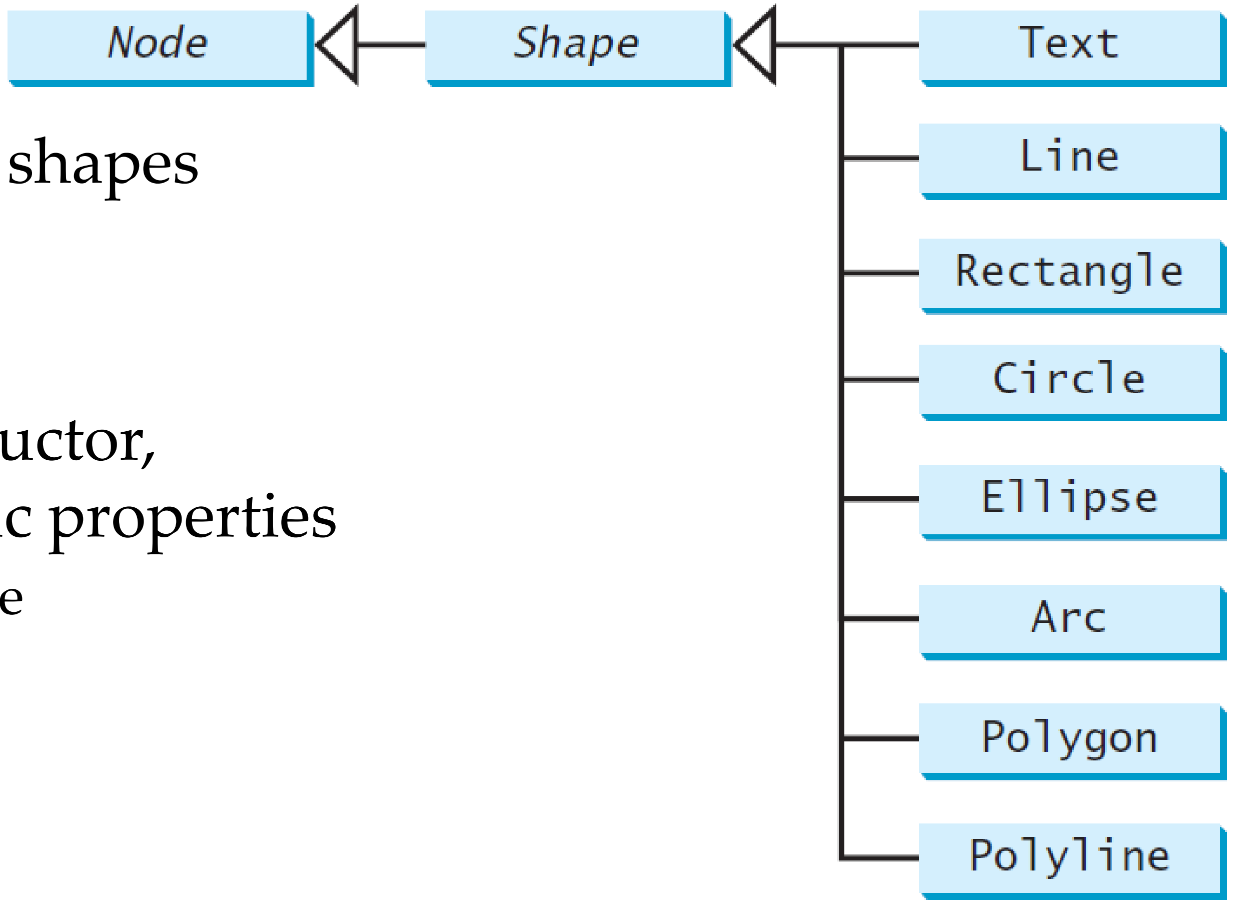
Sjaak re-enter "admin" to confirm.

heading	
name	nameField
password	pwdField
	btn
feedback	

DRAWING YOUR OWN ELEMENTS: SHAPES

shapes

JavaFX has many classes for specific shapes



`Shape` offers methods for all shapes

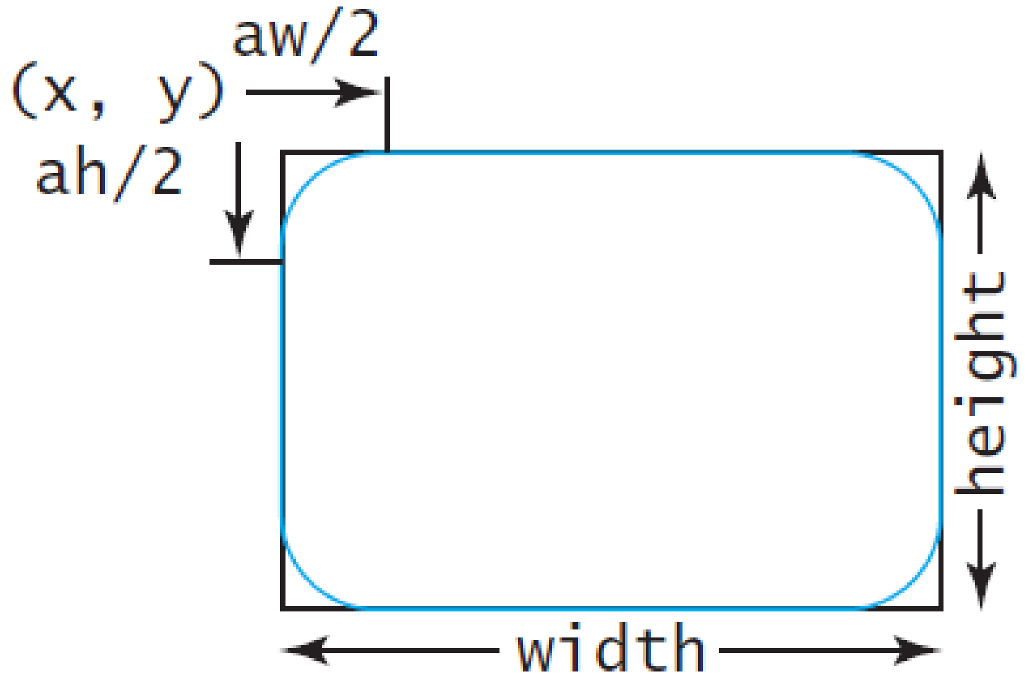
- fill color
- …

each class has its own constructor,
getters and setters for specific properties

- width and height of a rectangle
- radius of a circle
- points of a polygon
- …

rectangle

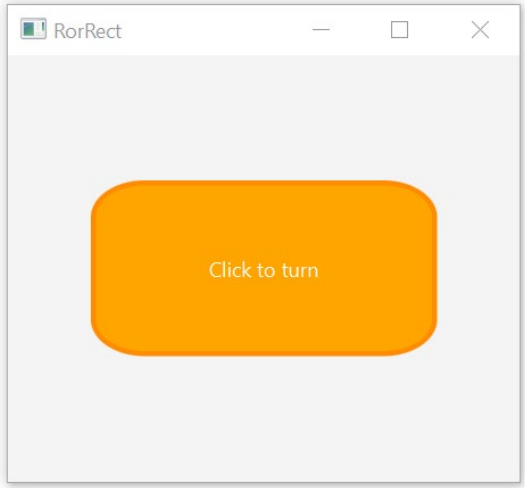
```
Rectangle r = new Rectangle(x, y, width, height);
r.setArcHeight(ah);
r.setArcWidth(aw);
```

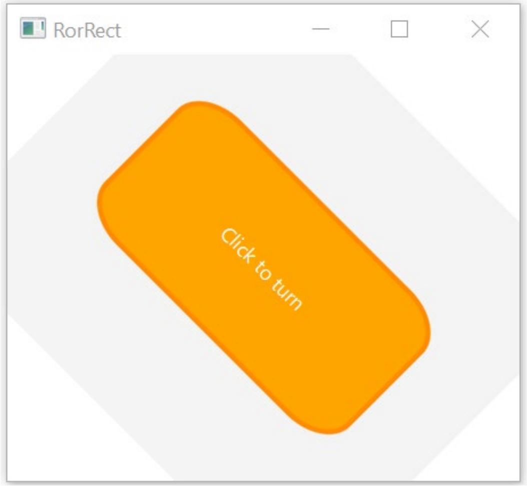


rectangle application

```
public void start(Stage primaryStage) {
  Rectangle rect = new Rectangle(200, 100);
  rect.setArcHeight(40);
  rect.setArcWidth(60);
  rect.setFill(Color.ORANGE);
  rect.setStroke(Color.DARKORANGE);
  rect.setStrokeWidth(3);
  Label label = new Label("Click to turn");
  label.setTextFill(Color.WHITE);
  StackPane root = new StackPane(rect, label);
  Scene scene = new Scene(root, 300, 250);
  root.setOnMouseClicked( e -> root.setRotate(root.getRotate() + 15));
  primaryStage.setTitle(this.getClass().getSimpleName());
  primaryStage.setScene(scene);
  primaryStage.show();
}
```

no x, y



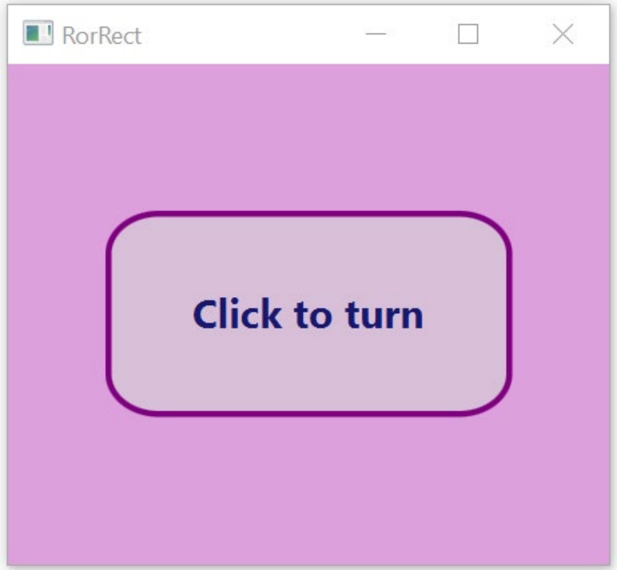


Adding a mouse click event handler turns pane into a "button"

rectangle application: using JavaFX CSS

setting all style options at once

```
public void start(Stage primaryStage) {
  Rectangle rect = new Rectangle(200, 100);
  rect.setStyle("-fx-fill: thistle; -fx-stroke: purple; -fx-stroke-width: 3; -fx-arc-height: 40; -fx-arc-width: 50");
  Label label = new Label("Click to turn");
  label.setStyle("-fx-text-fill: midnightblue;  -fx-font-size: 20; -fx-font-weight: bold");
  StackPane root = new StackPane(rect, label);
  root.setStyle("-fx-background-color: plum;");
  Scene scene = new Scene(root, 300, 250);
  root.setOnMouseClicked( e -> root.setRotate(root.getRotate() + 15));
  primaryStage.setTitle(this.getClass().getSimpleName());
  primaryStage.setScene(scene);
  primaryStage.show();
}
```

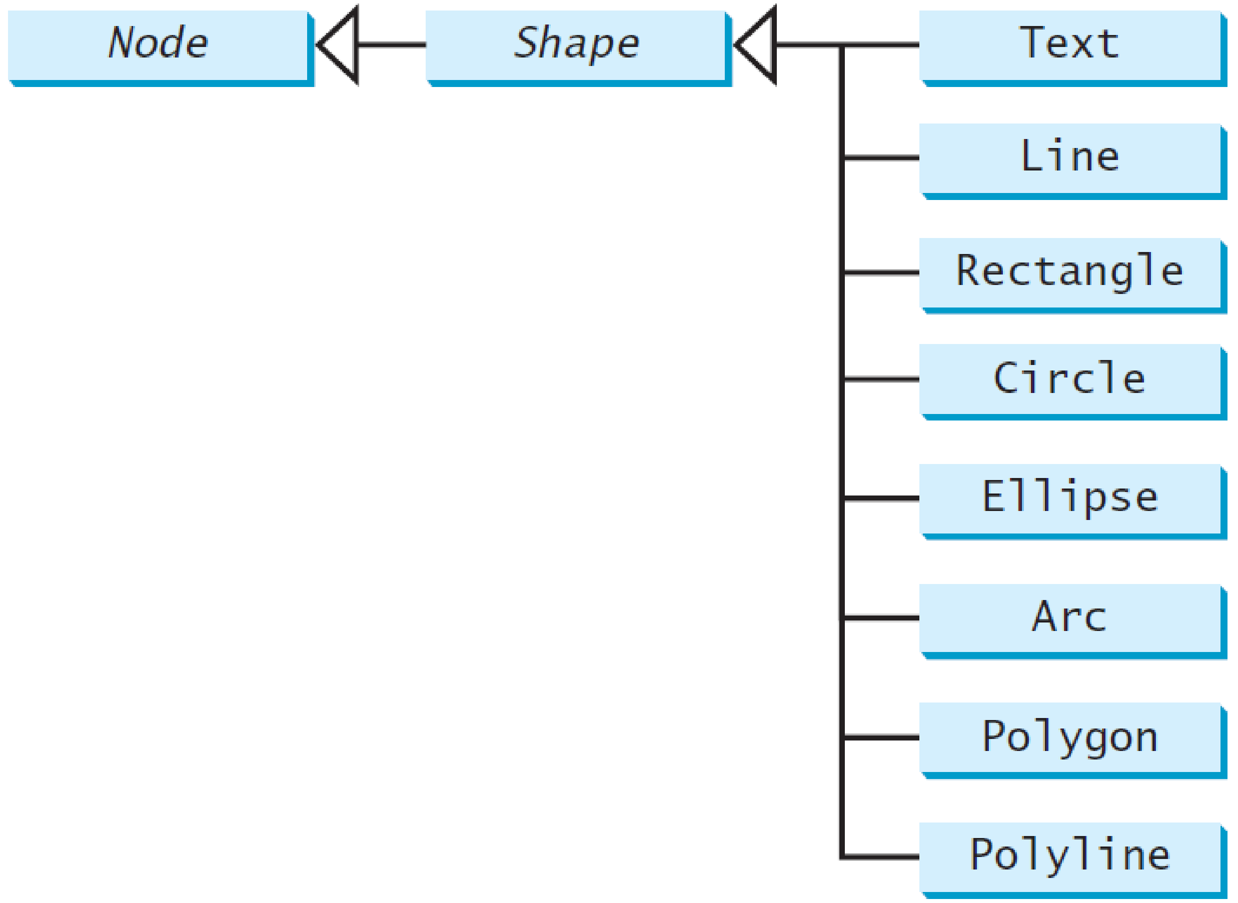


stroke, strokeWidth

stroke is the line surrounding a shape

shape has methods to set stroke properties: color, width, dash, lineCap, type, ..

works for (almost) any shape



polygon , polyline

pairs of (x, y) points

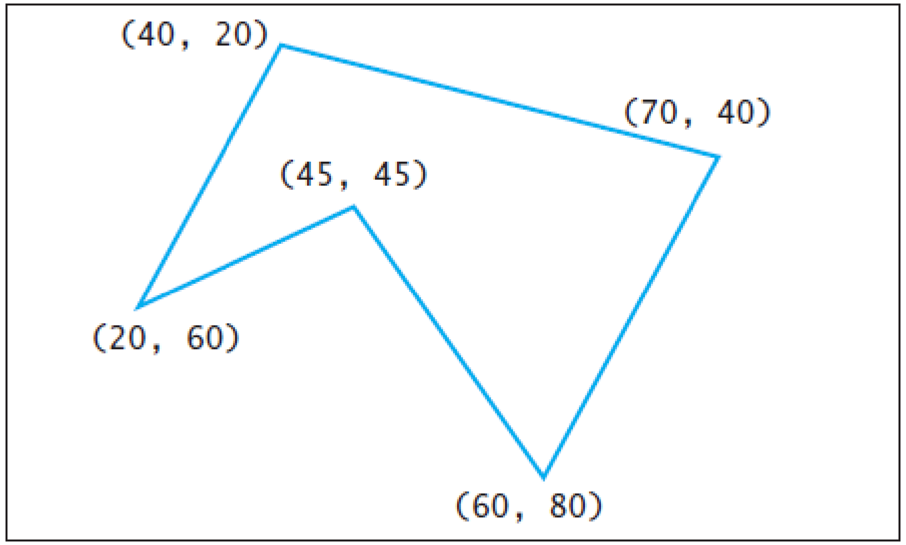
polygon is closed by the system

- automatic line segment from finish to start

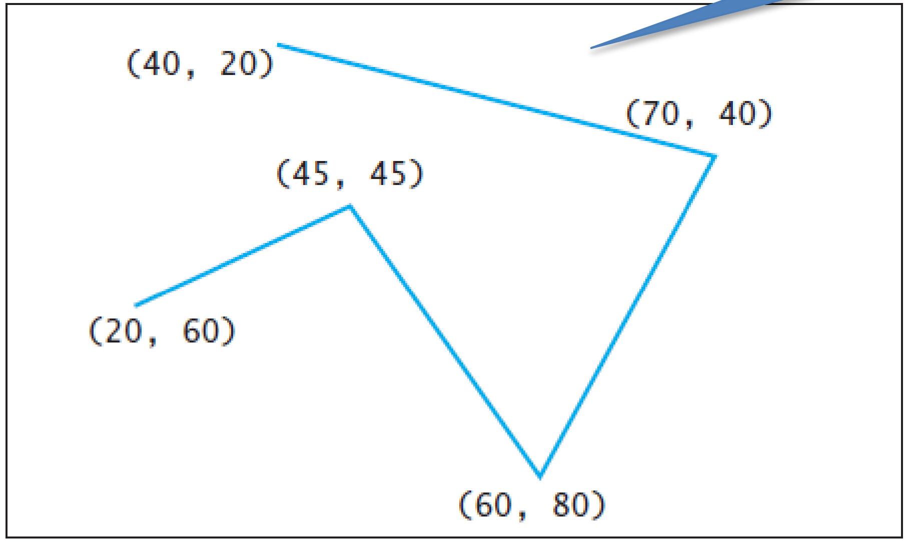
```
Polygon p = new Polygon(x1, y1, x2, y2, …);
```

use even number of arguments! (x,y) for each point!

not closed



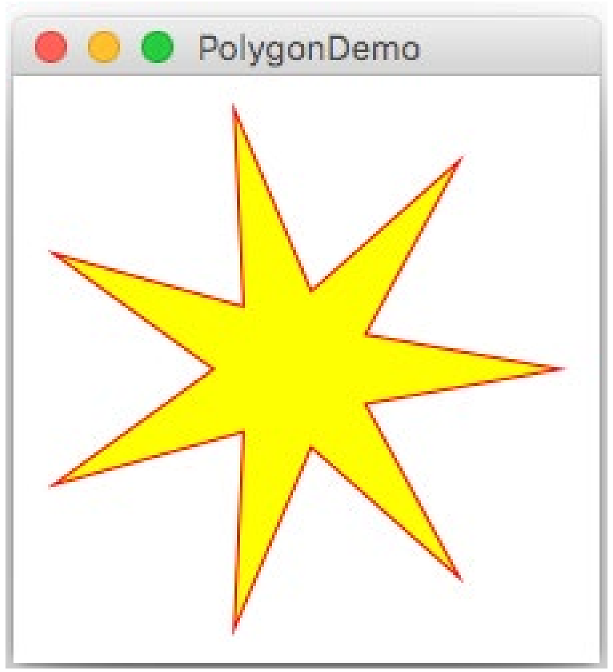
(a) Polygon



(b) Polyline

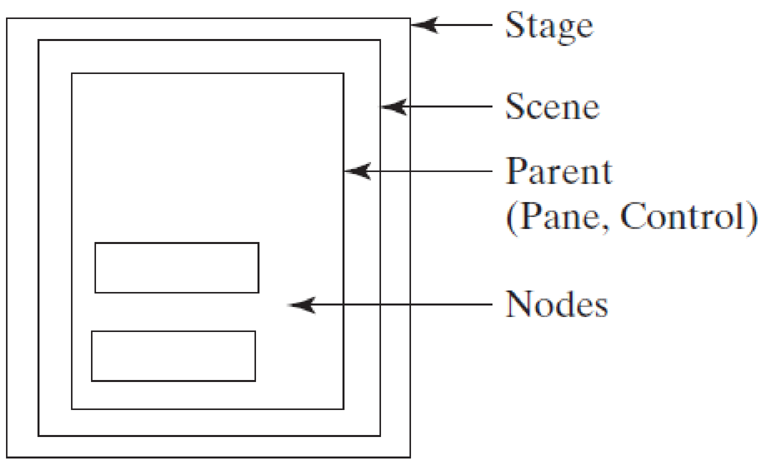
polygon demo

```
public class PolygonDemo extends Application {
  private final int N = 14;
  public void start(Stage stage) {
    Pane root = new StackPane();
    Polygon poly = new Polygon();
    root.getChildren().add(poly);
    poly.setFill(Color.YELLOW);
    poly.setStroke(Color.RED);
    ObservableList<Double> list = poly.getPoints();
    for (int i = 0; i < N; i += 1) {
      int r = (i % 2 == 0) ? 100 : 30;
      list.add(r * Math.cos(2 * Math.PI * i / N));
      list.add(r * Math.sin(2 * Math.PI * i / N));
    }
    Scene scene = new Scene(root, 220, 220);
    stage.setTitle(this.getClass().getSimpleName());
    stage.setScene(scene);
    stage.show();
  }
```

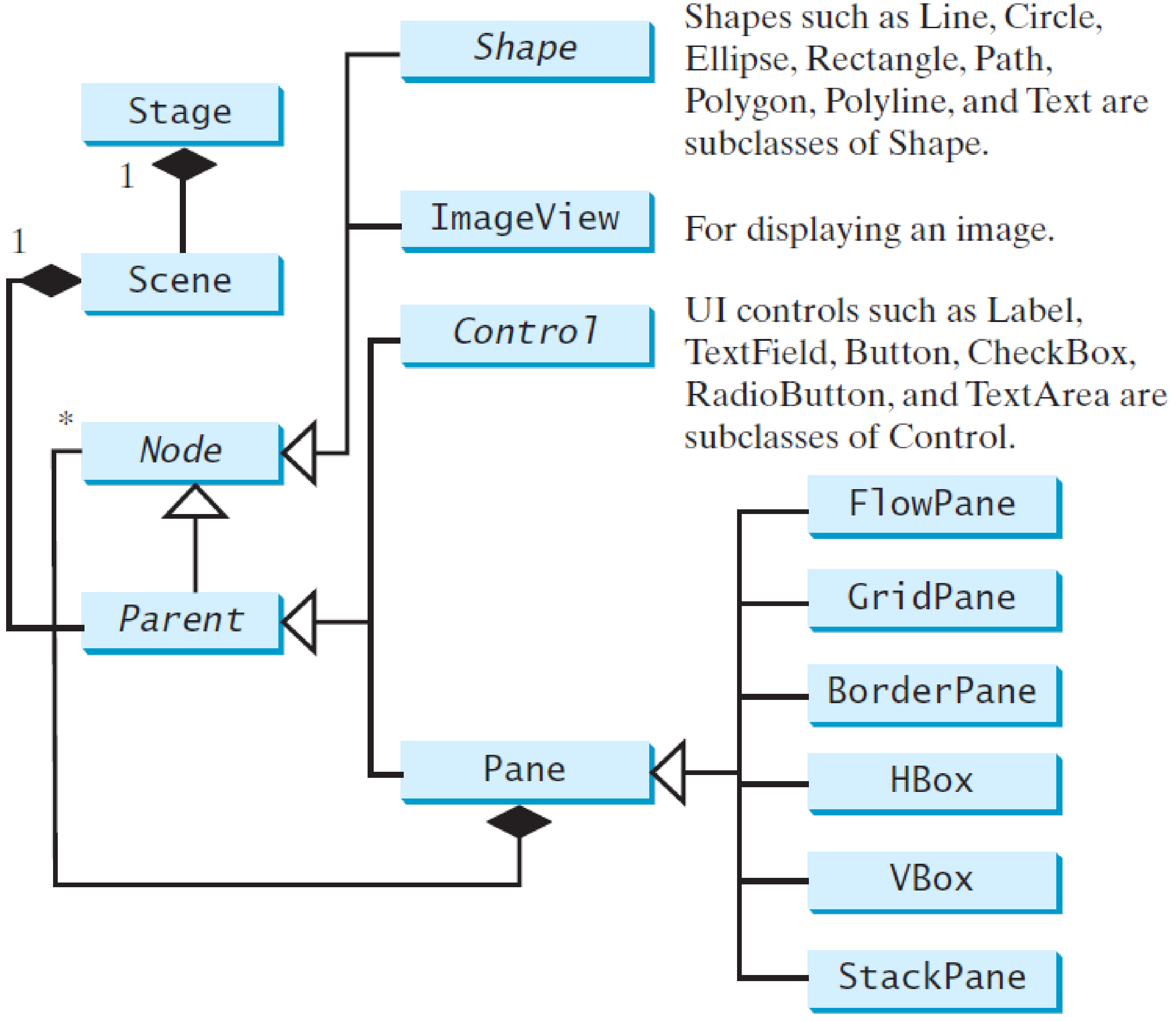


recap

- make subclass of Application
- launch instead of a constructor



(a)



Stage
1
Scene
1
*
Node
Parent
Shape
ImageView
Control
Pane
FlowPane
GridPane
BorderPane
HBox
VBox
StackPane
Shapes such as Line, Circle, Ellipse, Rectangle, Path, 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.

(b)

Lecture 9: GUIs: JavaFX (II)