GUIs: Input & Time

Object-Oriented Programming Lecture 9

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

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

April 13 , 2021

Radboud University

HANDLING THE MOUSE

mouse handling

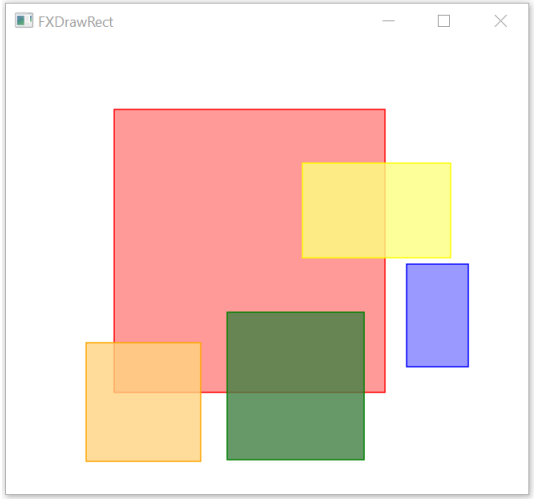
MouseEvents are fired whenever a mouse button is

- pressed
- released
- clicked
- moved
- dragged

on a node or a scene, and when the cursor enters / exits a node.

Read the text on
https://docs.oracle.com/javase/8/javafx/api/javafx/scene/input/MouseEvent.html
for details on mouse events, and drag & drop (though you shouldn't need that)

mouse handling



there can be more than one node under the mouse
each of them can have a mouse handler

who is handling the mouse event (first)?

target of event is selected when mouse button is pressed

all subsequent events are delivered to the same target

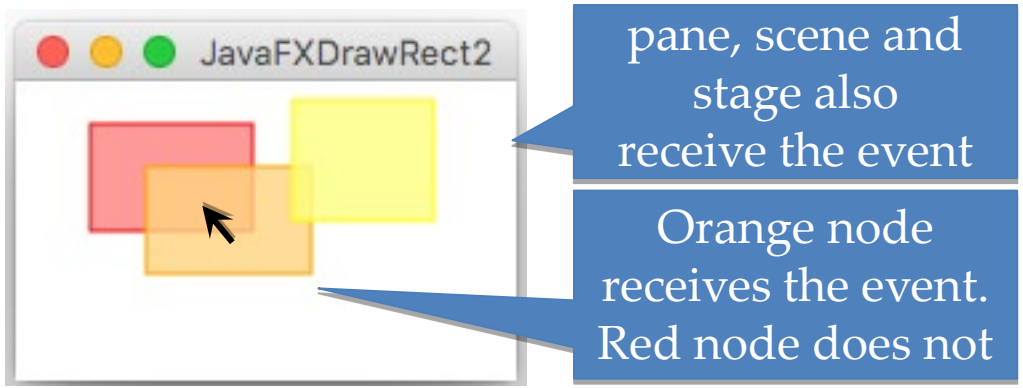
- until the button is released
- hence you can drag behind another node !

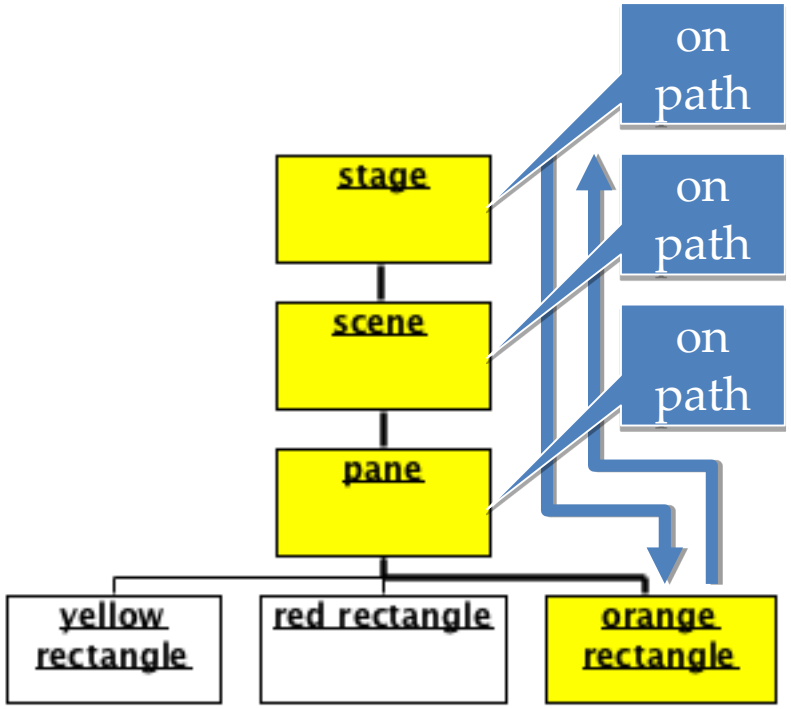
mouse pointer's location:

- `x, y`: relative to the origin `(0, 0)` of the `MouseEvent`'s node
- `sceneX, sceneY`: relative to origin of the Scene that contains the node
- `screenX, screenY`: relative to origin of screen that contains the mouse

JavaFX event delivery ("dispatching")

1. target selection
   - the target is the node at the location of the mouse
   - by default only the top node at this level receives the event
2. route construction (`EventDispatchChain`)
   - path from stage to node
3. event capturing
   - pass event top-down, apply **filters** on the path; filters can stop or redirect the event handling
4. event bubbling
   - invoke **handlers** bottom-up until it is consumed





mouse event handlers

each object on the route can have its own handler(s)

- without handler calling `event.consume()` the event is just passed further up

there can be handlers for many gestures:

- `setOnMouseClicked`, quick pressed and release on same node
  `setOnMousePressed`,
  `setOnMouseReleased`, only when mouse pressed on this node
  `setOnMouseDragged` only when mouse pressed on this node
  `setOnMouseDragEntered`, `setOnMouseDragExited`, `setOnMouseDragOver`, `setOnMouseDragReleased`, `setOnMouseEntered`, `setOnMouseExited`, `setOnMouseMoved`
- these are setters for **Properties** that associate a single handler to their event type

can register multiple handlers to the same event

- `addEventHandler(EventType, EventHandler)`
- `removeEventHandler(EventType, EventHandler)`

«interface» *EventHandler<T extends Event>*
`+handle(event: T)`

mouse event filters

each object on the route can have its own filter(s)

- without filter calling `event.consume()` the event is passed down (normal behaviour)

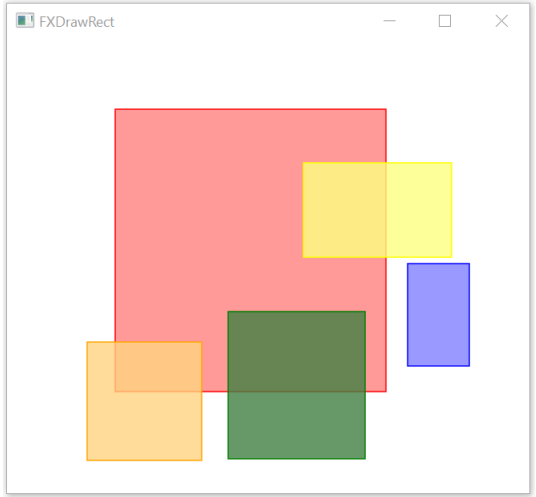
filters are implementations of `EventHandler`!

- the only difference with handlers is that filters are called during event **capturing**, handlers during event **bubbling**

can register multiple filters to the same event

- `addEventFilter(EventType, EventHandler)`
- `removeEventFilter(EventType, EventHandler)`

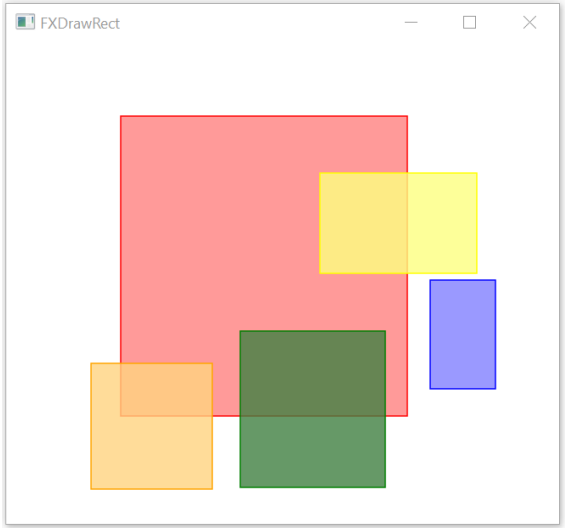
mouse events: drawing rectangles




```
public class FXDrawRect extends Application {
  private static final double MIN_SIZE = 5;
  private static final Color[] colors = {Color.RED,…};
  private int currentColorIx;
  private Rectangle currentRect;
  private double currentXOffset, currentYOffset;
  @Override
  public void start(Stage stage){
    Pane pane = new Pane();
    pane.setOnMousePressed(e -> newRect(pane, e));
    pane.setOnMouseDragged(e -> dragRect(e));
    pane.setOnMouseReleased(e -> finishRect(e));
    Scene scene = new Scene(pane, 300, 250);
    stage.setTitle(this.getClass().getSimpleName());
    stage.setScene(scene);
    stage.show();
  }
```

state for mouse dragging: store rect for creating, and offset for dragging around

creating a rectangle

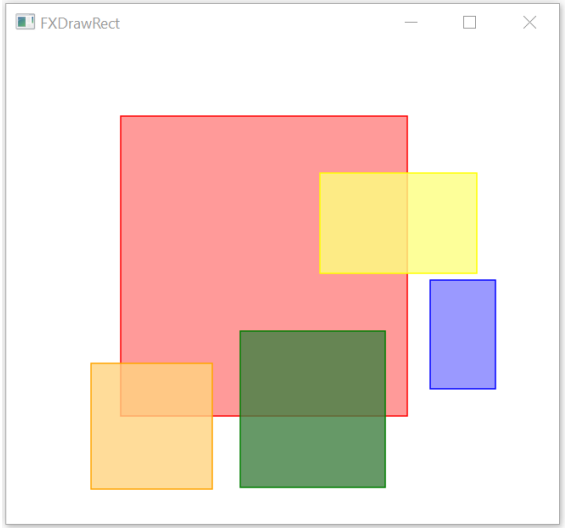


make a new rectangle at the position of the mouse
attribute `currentRect` is used as a reference to this new object

```
private void newRect (MouseEvent e) {
  currentRect = new Rectangle(e.getX(), e.getY(), MIN_SIZE, MIN_SIZE);
  Color nextColor = colors[currentColorIx];
  currentRect.setFill(nextColor.deriveColor(1, 0.5, 1, 0.8));
  currentRect.setStroke(nextColor);
  currentColorIx = (currentColorIx + 1) % colors.length;
  pane.getChildren().add(currentRect);
}
```

hueShift,
saturationFactor,
brightnessFactor,
opacityFactor

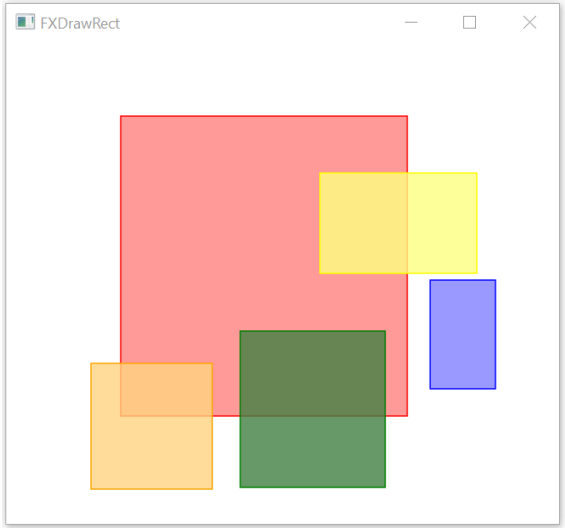
dragging a rectangle



resize the newly created rectangle

```
private void dragSizeRect(MouseEvent e) {
  double newWidth  = max(e.getX() - currentRect.getX(), MIN_SIZE);
  double newHeight = max(e.getY() - currentRect.getY(), MIN_SIZE);
  currentRect.setWidth(newWidth);
  currentRect.setHeight(newHeight);
}
```

finish a rectangle




```
private void finishRect (MouseEvent e) {
  Rectangle thisRect = currentRect;
  thisRect.setOnMousePressed(e2 -> {
    if (e2.isShiftDown()) {
        thisRect.toFront();
    }
    currentXOffset = e2.getX() - thisRect.getX();
    currentYOffset = e2.getY() - thisRect.getY();
    e2.consume(); });
 thisRect.setOnMouseDragged(e2 -> {
   thisRect.setX(e2.getX() - currentXOffset);
   thisRect.setY(e2.getY() - currentYOffset);
   e2.consume(); });
}
```

handlers for this rectangle

check whether modifier key was down

stop propagating event up

stop propagating event up

HANDLING THE KEYBOARD

keyboard handling

KeyEvents are fired whenever a keyboard key is

- typed
- pressed
- released

higher level, only unicode characters

lower level, using "key codes"

on a **focused** node or scene.

See https://docs.oracle.com/javase/8/javafx/api/javafx/scene/input/KeyCode.html for documentation of key codes

Modifier keys (ctrl, alt, meta, shift) tracked in KeyEvents
and generate key press / release events

Read the text on
https://docs.oracle.com/javase/8/javafx/api/javafx/scene/input/KeyEvent.html
for details on key events

key event handlers

no route like with `MouseEvent`, `KeyEvent` goes to **focused** element

- call `Node` method `requestFocus()` to request the focus
- there are restrictions, read the Javadoc of `requestFocus()`!

there can be handlers for many gestures:

- `setOnKeyTyped`, when a unicode character (e.g. s, r, y, å, ĉ, ş …) is typed
- `setOnKeyPressed`, when a key is pressed down
  `setOnKeyReleased` when a key is released
- these are setters of **Properties** that associate a single handler to their event type

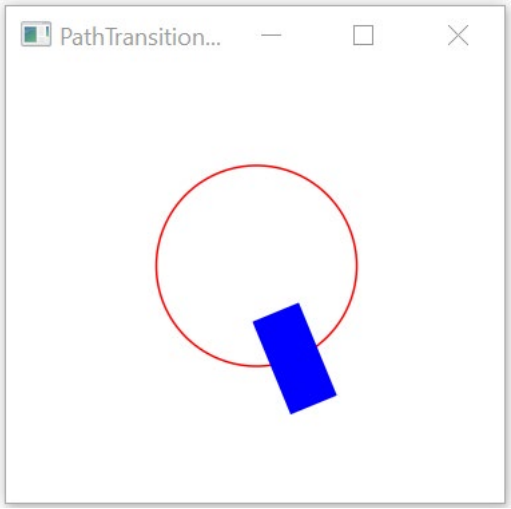
DOING STUFF WITHOUT INPUT: TIME

time

various options in JavaFX:

1. `Transitions`: predefined `Animations`
   - path transition: node follows a path (defined by JavaFX `Shape`)
   - fade transition: fading a node
   - {rotate, fill, scale, stroke, translate, pause}transition
   - parallel transitions: transitions at the same time
   - sequential transitions: one after an other
2. `Timeline`: property of node is changed in time
   - 'any' transformation, using `KeyFrames`
3. `Timeline` update handlers: explicitly set attributes after elapsed time
   - explicit handling of `TimeLine` updates in your program

Transition




```
public void start(Stage stage) {
  ... // create rectangle, circle, and pane.
  pane.getChildren().addAll(circle, rect);
  PathTransition pt = new PathTransition();
  pt.setPath(circle);
  pt.setNode(rect);
  pt.setDuration(Duration.seconds(4));
  pt.setOrientation(PathTransition.OrientationType.ORTHOGONAL_TO_TANGENT);
  pt.setCycleCount(Timeline.INDEFINITE);
  pt.setAutoReverse(true);
  pt.play();
  pane.setOnMouseEntered(e -> pt.pause());
  pane.setOnMouseExited(e -> pt.play());
  ... // add pane to scene, show scene in stage.
}
```

`TimeLine`

`pathTransition`, `fadeTransition`, `rotateTransition`, etc. predefine common transitions

`TimeLine` can be used to define any transition

- based on `keyFrames`
- e.g. `new TimeLine(KeyFrame… keyFrames)`
- the `keyFrames` are executed sequentially
- timeLines often use node properties

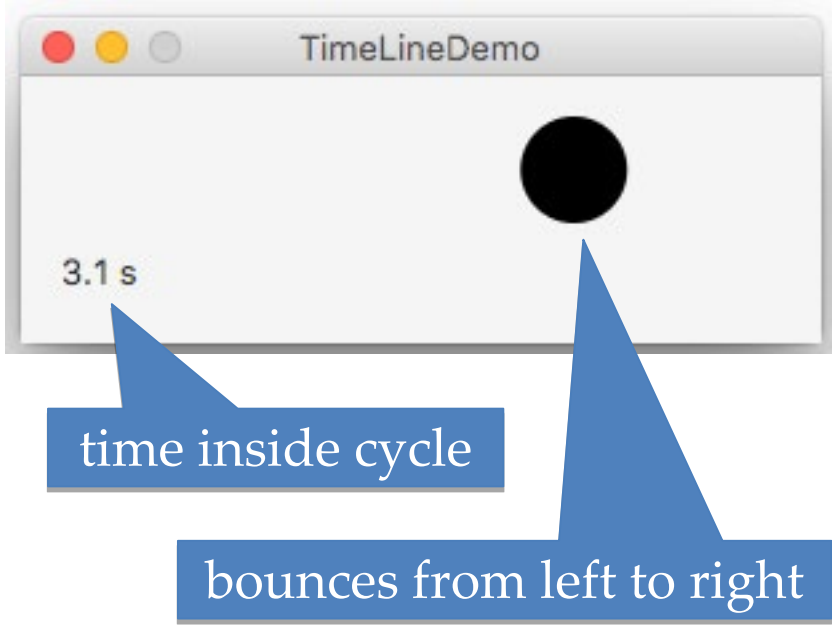
a `KeyFrame` gives target values at points in time for a set of variables that are interpolated on a `Timeline`

- e.g. `KeyFrame(Duration time, KeyValue... values)`

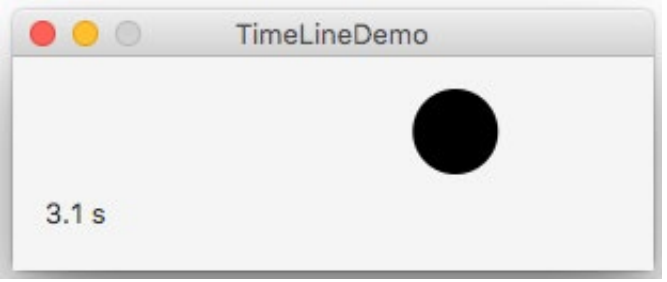
`KeyValue` is an immutable class for points in a `KeyFrame`

TimeLine with KeyFrame updating property

```
public void start(Stage stage) {
  final int R = 20;
  stage.setTitle(getClass().getSimpleName());
  stage.setResizable(false);
  Circle circle = new Circle(2 * R, 2 * R, R);
  Label text = new Label();
  VBox vbox = new VBox(10);
  vbox.setPadding(new Insets(15));
  vbox.getChildren().addAll(circle, text);
  Scene scene = new Scene(vbox, 300, 100);
  stage.setScene(scene);
  stage.show();
  ...
```



TimeLine with KeyFrame updating property




```
Timeline timeline = new Timeline();
KeyValue kv =
    new KeyValue(circle.translateXProperty(), scene.getWidth() - 2 * R);
KeyFrame keyFrame = new KeyFrame(Duration.seconds(4), kv);
timeline.getKeyFrames().add(keyFrame);
timeline.setCycleCount(4);
timeline.setAutoReverse(true);
timeline.currentTimeProperty().addListener((Observable ov)
    -> text.setText(String.format("%.1f s",
            timeline.getCurrentTime().toSeconds()))
);
timeline.setOnFinished((ActionEvent ae) -> stage.close());
timeline.play();
}
```

property to be changed

target value

time

timeLine handler

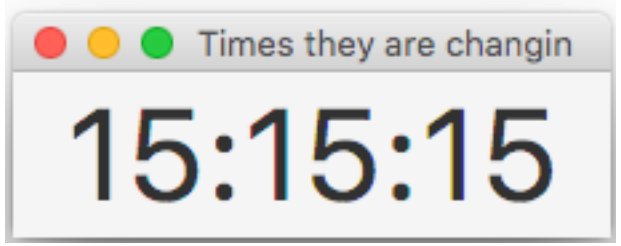
here we have a KeyFrame and a handler, often only one of them

don't forget this

TimeLine with KeyFrame calling a handler: a digital clock

```
public void start(Stage primaryStage) {
  timeLabel = new Label();
  timeLabel.setFont(Font.font(48));
  StackPane root = new StackPane();
  root.getChildren().add(timeLabel);
  ... // create scene, show stage.
  setTime();
  Timeline timeline = new Timeline(new KeyFrame(Duration.seconds(1), e->setTime()));
  timeline.setCycleCount(Timeline.INDEFINITE);
  timeline.play();
}
private void setTime() {
  LocalTime now = LocalTime.now();
  String time =
      String.format("%d:%02d:%02d",now.getHour(),now.getMinute(),now.getSecond());
  timeLabel.setText(time);
}
```

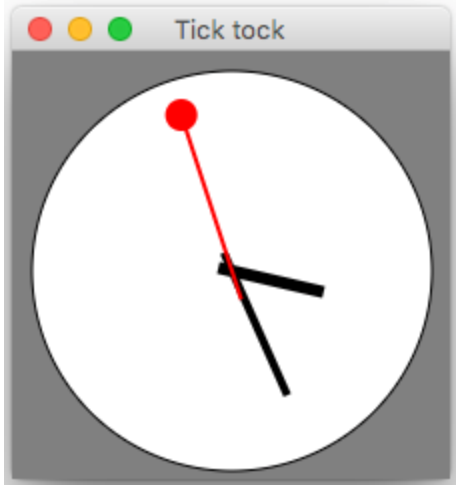
no KeyValue !



TimeLine with **transformations**: an analog clock

radius of circle, midpoint X and midpoint Y in pane

```
protected final int R = 100, MX = R + 10, MY = R + 10;
private Rotate hourRot, minRot, secRot;

public void start(Stage stage) {
  Pane pane = new Pane();
  Circle circle = new Circle(MX, MY, R);
  circle.setFill(Color.WHITE);
  circle.setStroke(Color.BLACK);

  Line hourHand = new Line(MX, MY + 4, MX, MY * 0.6);
  hourHand.setStrokeWidth(6);
  hourRot = new Rotate(0, MX, MY);
  hourHand.getTransforms().add(hourRot);

  Line minHand = new Line(MX, MY + 8, MX, MY * 0.4);
  minHand.setStrokeWidth(4);
  minRot = new Rotate(0, MX, MY);
  minHand.getTransforms().add(minRot);
```



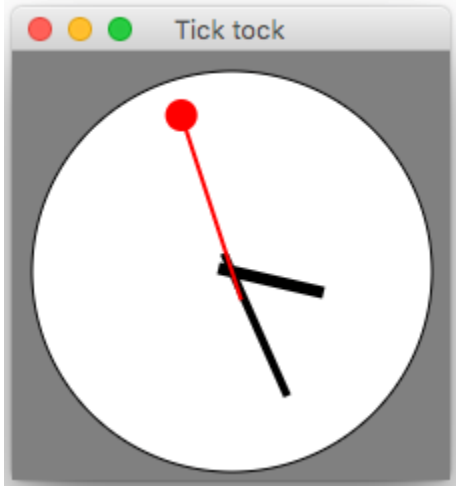
rotation angle 0, around a pivot (MX, MY)

a list of transformations

much easier than computing end point with sin and cos

`TimeLine` with **transformations**: an analog clock

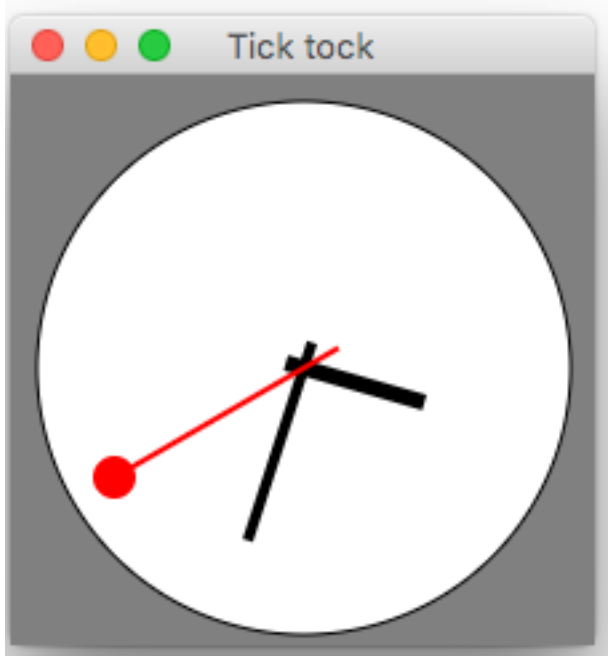
```
  Line secLine = new Line(MX, MY + 14, MX, MY * 0.2);
  secLine.setStrokeWidth(2);
  secLine.setStroke(Color.RED);
  Circle secCircle = new Circle(MX, MY - R * 0.82, 8);
  secCircle.setFill(Color.RED);
  Group secHand = new Group();
  secHand.getChildren().addAll(secLine, secCircle);
  secRot = new Rotate(0, MX, MY);
  secHand.getTransforms().add(secRot);

  pane.getChildren().addAll(circle, hourHand, minHand, secHand);
  Scene scene = new Scene(pane, 2 * R + 20, 2 * R + 20);
  scene.setFill(Color.GRAY);
  stage.setTitle("Tick tock");
  stage.setScene(scene);
  stage.show();

  Timeline timeline = new Timeline(new KeyFrame( Duration.seconds(1), e->setTime()));
  timeline.setCycleCount(Animation.INDEFINITE);
  timeline.play();
  setTime();
}
```



`TimeLine` with **transformations**: an analog clock

time handler only changes the rotates:

```
private void setTime() {
  LocalTime now = LocalTime.now();
  hourRot.setAngle(now.getHour()   * 360 / 12 +
                   now.getMinute() * 360 / (60 * 12));
  minRot.setAngle( now.getMinute() * 360 / 60);
  secRot.setAngle( now.getSecond() * 360 / 60);
}
```



we could also change the rotates with `KeyValues` without using `KeyFrame` handlers!

A programmer, a surgeon, a prostitute and an engineer are having an argument about who has the world's oldest profession.

The prostitute opens the debate by saying: "My profession is nowadays always referred to as the world's oldest profession!"

The surgeon counters: "Ah, but historically, you needed both genders for socially accepted prostitution, and according to the bible god created Eve by taking a rib from Adam, which is surgery! So mine is the oldest profession!"

The engineer thinks they can win the argument: "Ah, but before your god even created Adam, he created the entire universe from chaos! That's what engineers do all the time, create things from chaos. So mine is definitely the oldest profession!"

But the programmer only chuckles and says:
"Who do you think was responsible for all the chaos?"

OBJECT-ORIENTED DESIGN: SEPARATING GUI AND DATA

plain JavaFX animation: bouncing balls

tight coupling between world objects and JavaFX objects

keyboard:

n: new ball

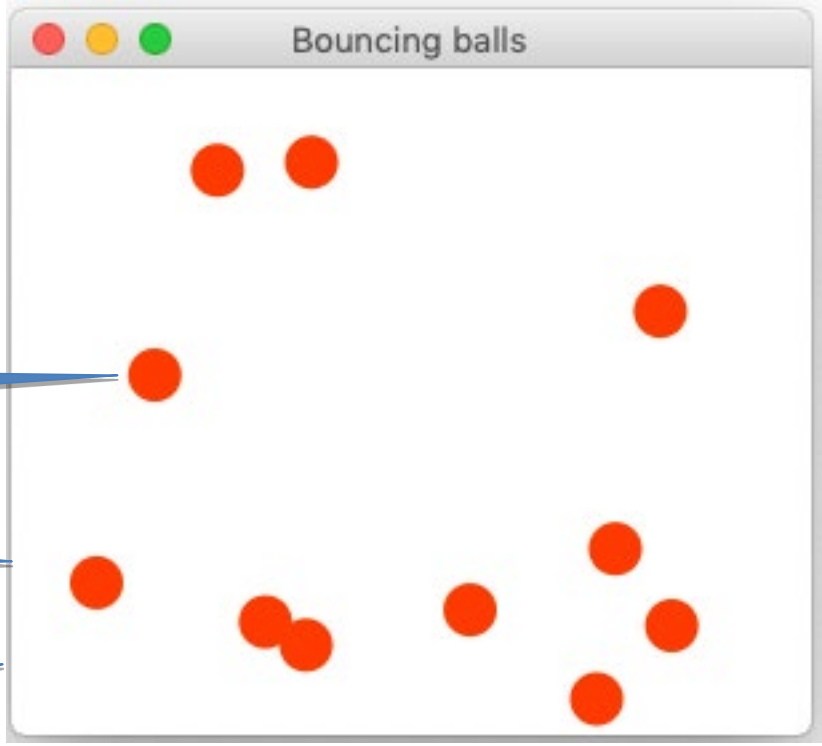
other: start/stop animation

timeline: move balls

Ball: subclass of Circle

World: subclass of Pane

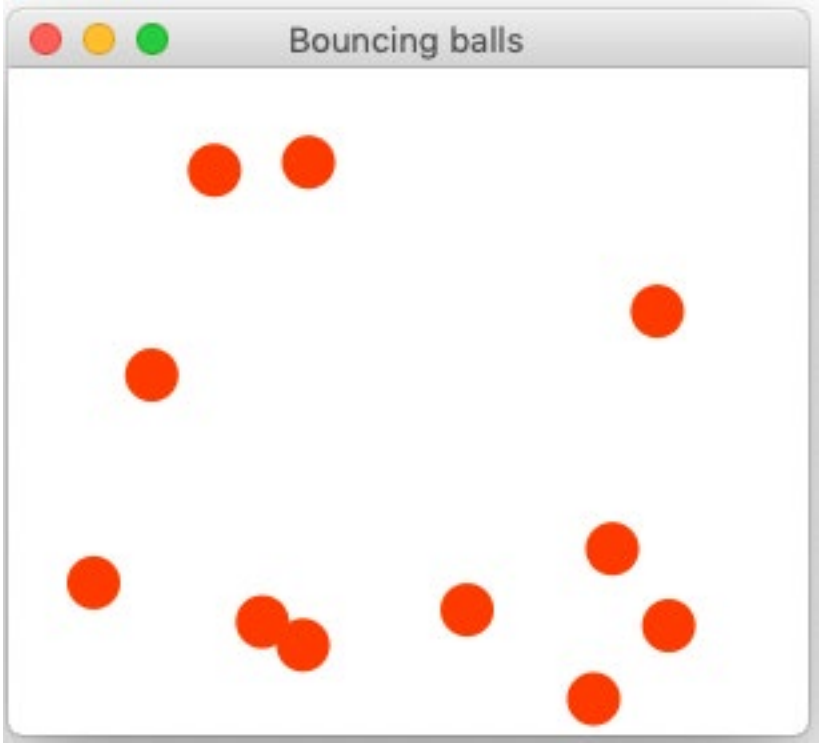
Scene: holds timeline and key handler



Ball 1 / 2

Ball is also its own JavaFX representation

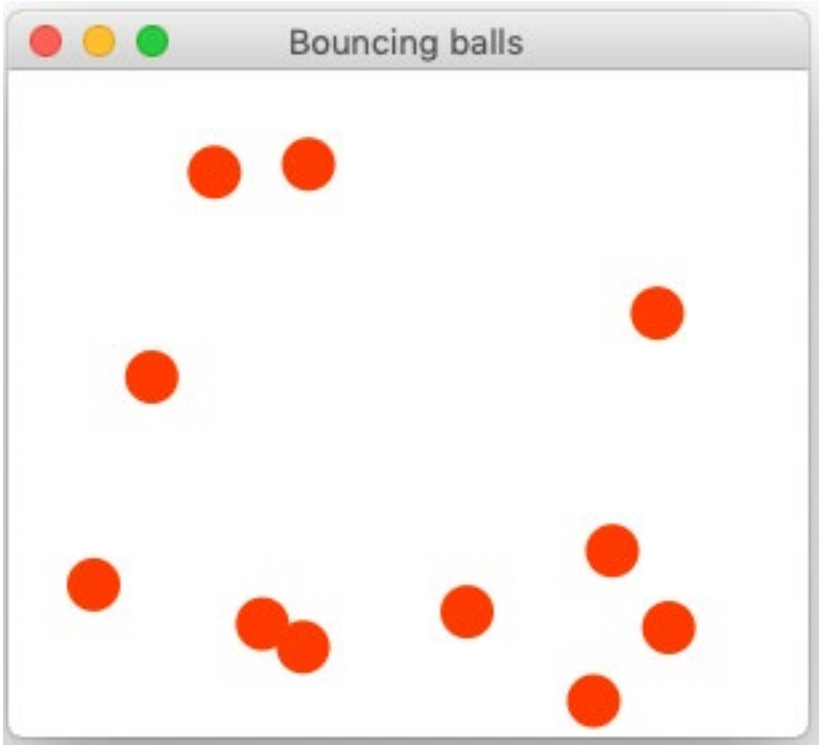
```
public class Ball extends Circle {
  private final static double R = 10;
  private double x = R;
  private double y = R;
  private double dx;
  private double dy;
  private Random random = new Random();

  Ball() {
    super(R, R, R, Color.ORANGERED);
    dx = 1 + 3 * random.nextDouble();
    dy = 1 + 3 * random.nextDouble();
  }
```



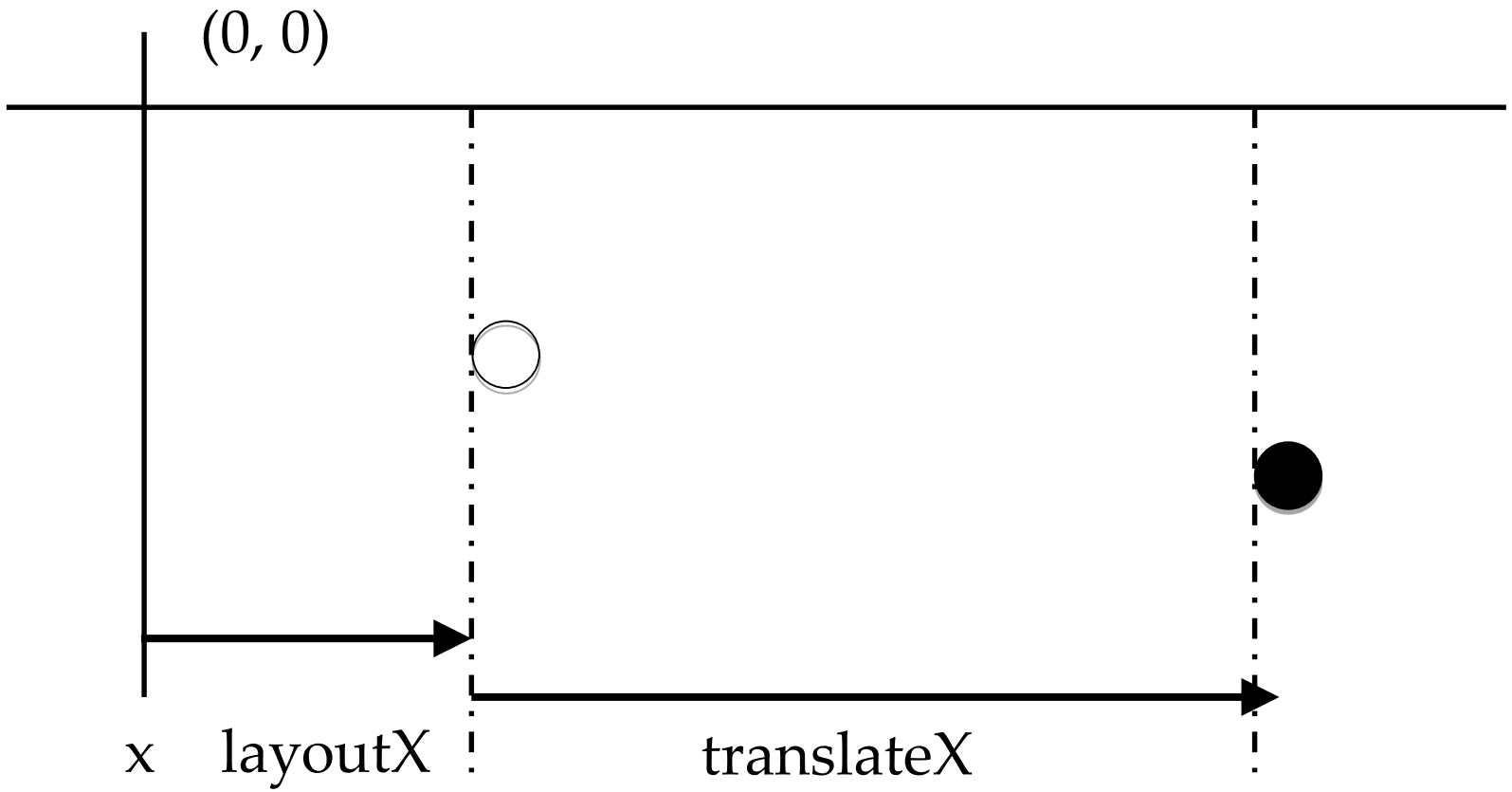
Ball 2 / 2

```
public void step(double maxX, double maxY) {
  x += dx;
  y += dy;
  if (x < R || x > maxX - R) {
    dx *= -1;
  }
  if (y < R || y > maxY - R ) {
    dy *= -1;
  }
  this.setTranslateX(x-getCenterX());
  this.setTranslateY(y-getCenterY());
}
```

bounce if outside world



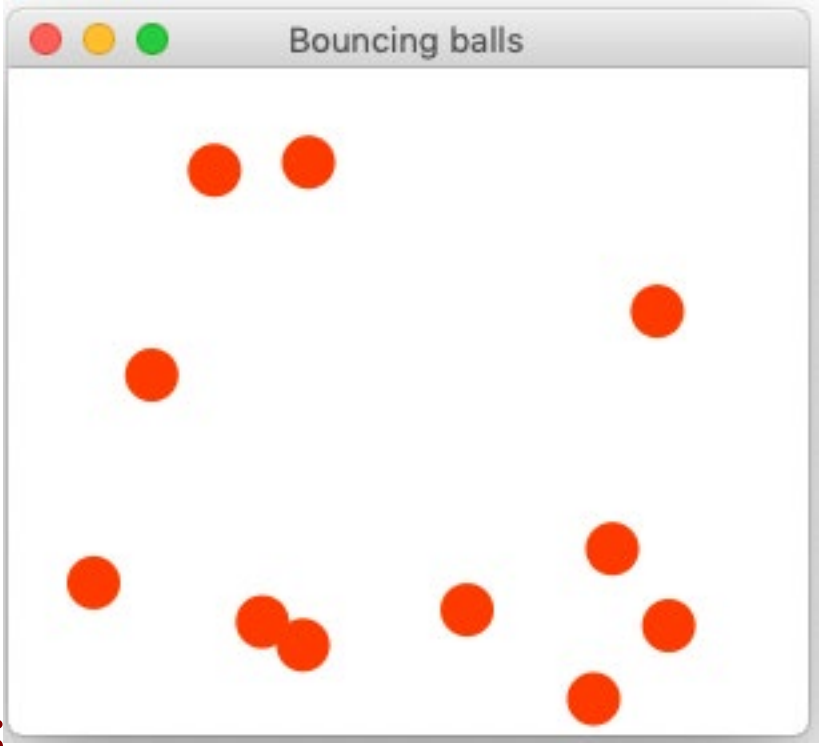
Node position



world

World is also its own JavaFX representation

```
public class World extends Pane {
  private List<Ball> balls = new ArrayList<>();
  public World () {
    super();
    newBall();
  }
  public void newBall() {
    Ball ball = new Ball();
    balls.add(ball);
    this.getChildren().add(ball);
  }
  public void tick() {
    for (Ball b : balls) {
      b.step(this.getWidth(), this.getHeight());
} } }
```

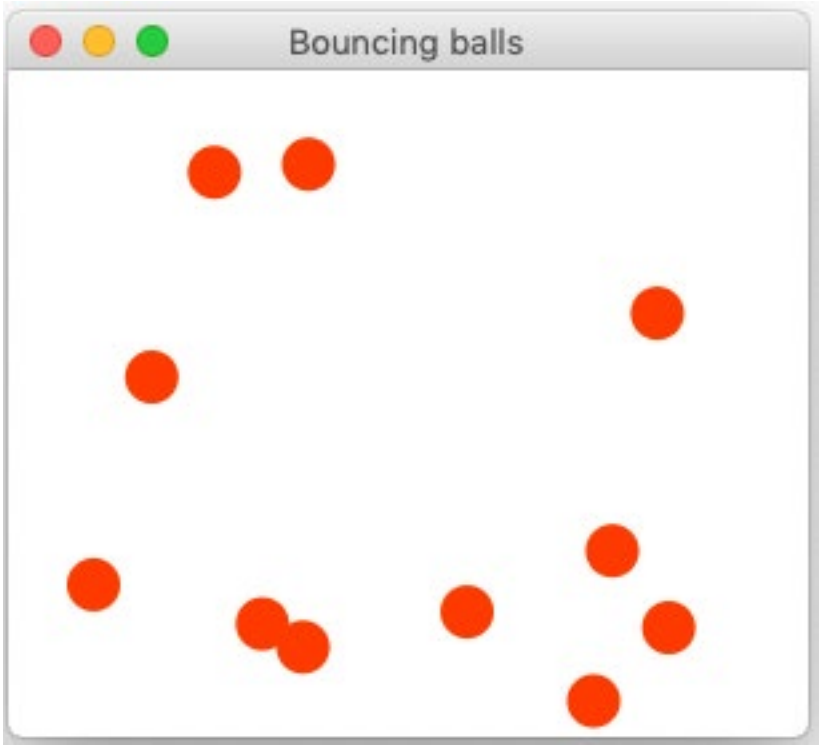


main 1 / 2

```
public class BounceBalls extends Application {
  private final static int DELAY = 100; // timer delay in ms
  private boolean running = true;
  @Override
  public void start(Stage primaryStage) {
    World world = new World();
    Timeline timeline
        = new Timeline(new KeyFrame( Duration.millis( DELAY ),
                       e -> world.tick() ));
    timeline.setCycleCount( Animation.INDEFINITE );
    timeline.play();
    Scene scene = new Scene(world, 300, 250);
```

main 2 / 2

```
  scene.setOnKeyPressed(event -> {
    if (event.getCode().equals(KeyCode.N)) {
      world.newBall();
      return;
    } else {
      if (running) {
        timeline.pause();
      } else {
        timeline.play();
      }
      running = !running;
    }
  });
  primaryStage.setTitle("Bouncing balls");
  primaryStage.setScene(scene);
  primaryStage.show();
}
```



bouncing balls review

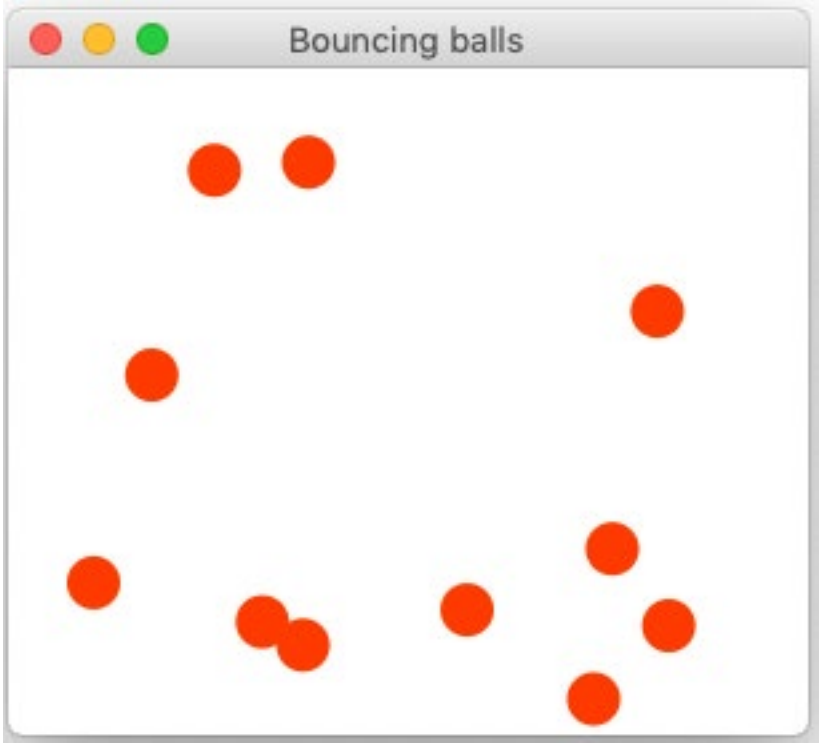
close coupling between state and its JavaFX view

```
public class Ball extends Circle { ..
public class World extends Pane { ..
```

this is convenient and perfect if the goal of our program is to draw something with JavaFX

otherwise it is essential to separate the state and its JavaFX view better

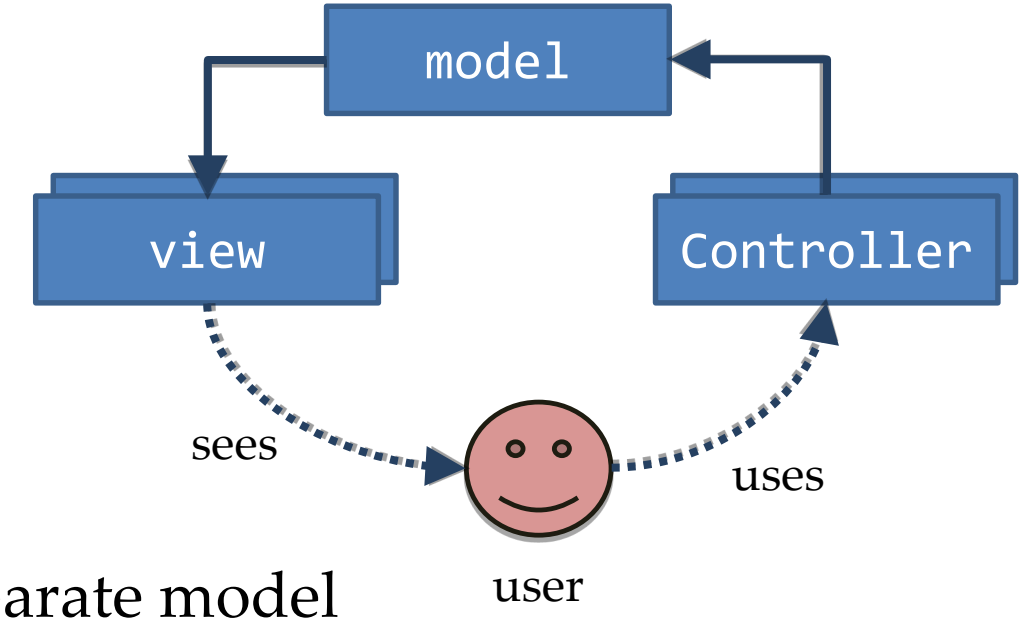
- like we told you in lecture 1
- the MVC pattern is one of the ways to do this



Model-View-Controller pattern

JavaFX can be structured according the MVC pattern: separate state and GUI

- Model: state of program
  - has no idea of GUI (views)
- View: the JavaFX stages displaying a view on the model
  - unknown by model
- Controller: the handlers changing the model



`Properties` are `ObservableValues`

- they can be (part of) the model

our examples focus on drawing (view) and handlers (controllers), there is usually no separate model

Properties, Observables, …

automatically updating a view if the model changes

OBSERVER PATTERN

observer pattern

model notifies views when something happens;
views attach themselves to model in order to be notified

button notifies action listeners when it is pressed;
action listeners attach themselves to button in order to be notified

generalize:
observers attach themselves to *subject* (i.e. the *observable*) ;
subject notifies *observers* when something happens to it

observer pattern

context

- one `Object`, the *subject*, is a source of events
- one or more observer `Objects` want to be notified when such an event occurs

solution

- define an Observer interface type,
  all concrete observers implement it
- the *subject* maintains a collection of observers
- the *subject* supplies methods for attaching (and detaching) observers
- whenever the state of the *subject* changes,
  the *subject* notifies all observers
- (the observers then query the subject for the new state)

bouncing ball as “properly” separated MVC

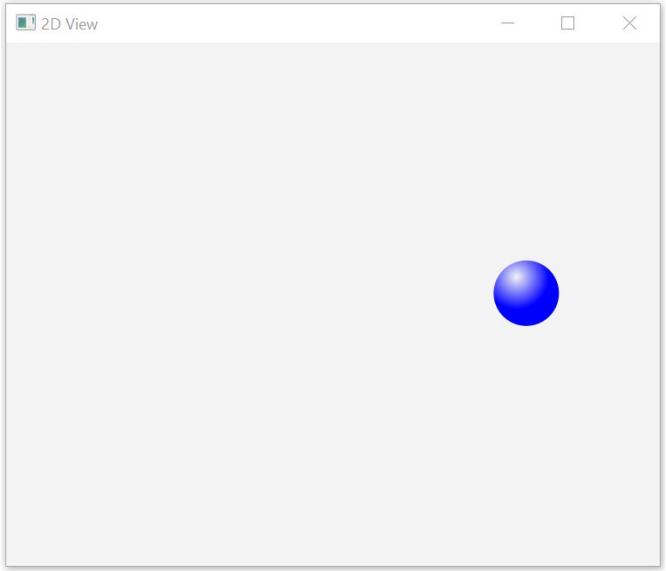
model: ball, world

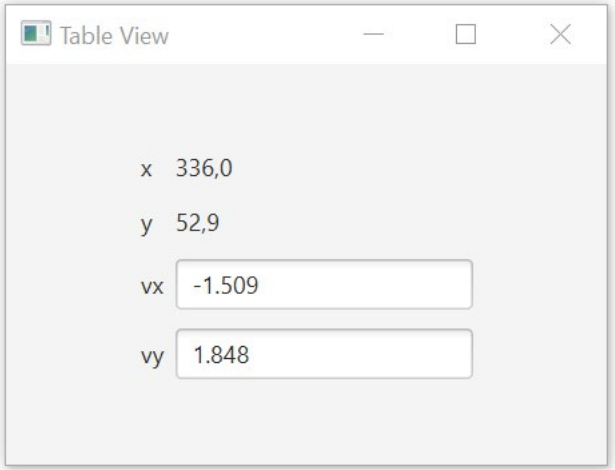
- xPos, yPos, xVel, yVel
- move

view 1: ball on canvas

- we can resize this window

view 2: ball position in table





controller 1: timer

controller 2: set velocity (i.e. displacement per unit of time)

- in same table as view 2

BallView: an observer interface

```
public interface BallView {
   void notify( Ball ball );
}
```

model class: Ball

```
public class Ball {
  public final static double RADIUS = 25;
  private double xPos, yPos;
  private DoubleProperty xVel, yVel;
  private World world;

  private List<BallView> views = new ArrayList<>();
  private Random random = new Random();

  public Ball( World world ) {
    this.world = world;
    xPos = RADIUS+(world.getWorldWidth()-2*RADIUS)*random.nextDouble();
    yPos = RADIUS+(world.getWorldHeight()-2*RADIUS)*random.nextDouble();
    xVel = new SimpleDoubleProperty(1 + 3 * random.nextDouble());
    yVel = new SimpleDoubleProperty(1 + 3 * random.nextDouble());
  }
```

list of observers

model class: Ball (continued)

```
  // getters omitted
  public void move() {
    xPos += xVel.doubleValue();
    if ( xPos < RADIUS || xPos > world.getWorldWidth() - RADIUS) {
      xVel.set(xVel.doubleValue() * -1);
    }
    yPos += yVel.doubleValue();
    if ( yPos < RADIUS || yPos > world.getWorldHeight() - RADIUS) {
      yVel.set(yVel.doubleValue() * -1);
    }
    notifyViews();
  }

  public void addView( BallView view ){
     views.add(view);
  }

  private void notifyViews(){
    for( BallView view: views) {
      view.notify(this);
    }
  }
}
```

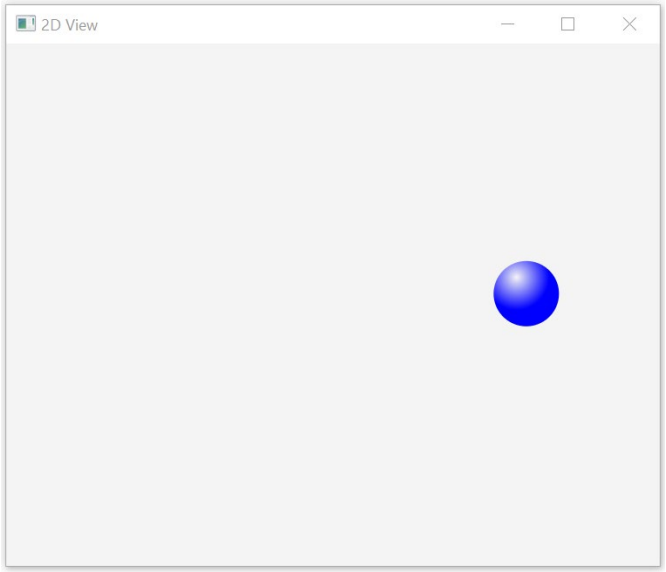
a view can register itself

model class: World

```
public class World {
  private double worldWidth, worldHeight;

  public World(double worldWidth, double worldHeight) {
    this.worldWidth = worldWidth;
    this.worldHeight = worldHeight;
  }

  // getters/setters omitted
}
```

BallView I:

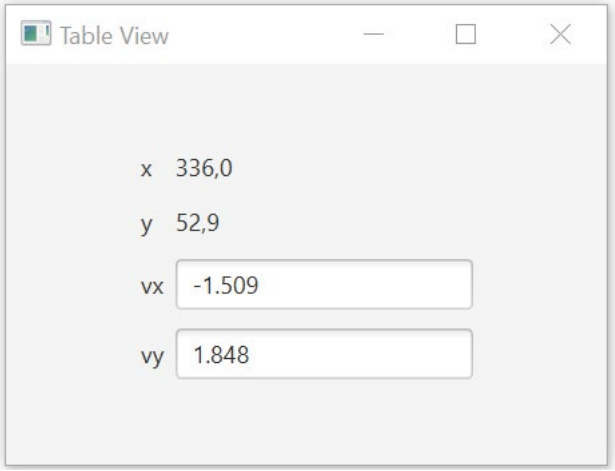
```
public class TwoDView extends Pane implements BallView {
  private Circle ball;
  public TwoDView(World world) {
    ball = new Circle(Ball.RADIUS);
    RadialGradient rg = new RadialGradient(
            0, 0, // focusAngle, focusDistance
            0.35, 0.25, 0.5, // centerX, centerY, radius
            true, CycleMethod.NO_CYCLE, // proportional, cycle
            new Stop(0.0, Color.WHITESMOKE), new Stop(1.0, Color.BLUE));
    ball.setFill(rg);
    getChildren().add(ball);
    widthProperty().addListener((ov, ow, nw) -> world.setWorldWidth(nw.doubleValue()));
    heightProperty().addListener((ov, ow, nw) -> world.setWorldHeight(nw.doubleValue()));
  }

  @Override
  public void notify(Ball b) {
    ball.setTranslateX(b.getXPos());
    ball.setTranslateY(b.getYPos());
  }
}
```



called after resize

BallView II : TableView

```
public class TableView extends GridPane implements BallView {
   private Label xLabel, yLabel;
   private TextField vxLabel, vyLabel;
   public TableView( Ball b ) {
      setAlignment(Pos.CENTER);
      setHgap(5);
      setVgap(10);
      add(new Label("x"), 0, 0);
      add(xLabel = new Label(""), 1, 0);
      add(new Label("y"), 0, 1);
      add(yLabel = new Label(""), 1, 1);
      add(new Label("vx"), 0, 2);
      add(vxLabel = new TextField(String.format("%1.1f",b.getXVel())), 1, 2);
      add(new Label("vy"), 0, 3);
      add(vyLabel = new TextField(String.format("%1.1f",b.getYVel())), 1, 3);
      vxLabel.textProperty().bindBidirectional(b.xVelProperty(), new NumberStringConverter());
      vyLabel.textProperty().bindBidirectional(b.yVelProperty(), new NumberStringConverter());
   }
   @Override
   public void notify(Ball ball) {
      xLabel.setText(String.format("%1.1f", ball.getXPos()));
      yLabel.setText(String.format("%1.1f", ball.getYPos()));
   }
}
```



bidirectional binding iso an observer to change the velocity

timer controller: Driver

```
public class Driver {
   private final Ball ball;

   public Driver(Ball ball) {
      this.ball = ball;
   }

   public void start() {
      Timeline loop = new Timeline(new KeyFrame(Duration.millis(10),
              e -> ball.move()));
      loop.setCycleCount(Timeline.INDEFINITE);
      loop.play();
   }
}
```

must know the model

tie everything together: BBMain

```
public class BBMain extends Application {
  final static double WIDTH_2D = 500, HEIGHT_2D = 400;
  final static double WIDTH_TV = 300, HEIGHT_TV = 200;
  @Override
  public void start(Stage stage) {
    World world  = new World(WIDTH_2D,HEIGHT_2D);
    Ball ball    = new Ball(world);
    TableView tv = new TableView(ball);
    TwoDView tdv = new TwoDView(world);
    ball.addView(tdv);
    ball.addView(tv);
    Driver driver = new Driver(ball);
    driver.start();
    Stage secondStage = new Stage();
    secondStage.setTitle("Table View");
    secondStage.setScene(new Scene(tv, WIDTH_TV, HEIGHT_TV));
    secondStage.show();
    stage.setTitle("2D View");
    stage.setScene(new Scene(tdv, WIDTH_2D, HEIGHT_2D));
    stage.show();
 }
```

you should be more than capable of looking these up,
reading about them,
and experimenting with them yourselves:

SOME MORE USEFUL WIDGETS

radio buttons 1/3

```
public class RadioDemo extends Application {
  private final String[] colors = {"Red", "Yellow", "Blue"};
  private final Button btn  = new Button("");
  private final CheckBox cb = new CheckBox("Border line");
  private final ToggleGroup group = new ToggleGroup();
  private final Circle circle = new Circle(50);
  @Override
  public void start(Stage stage) {
    VBox buttons = new VBox(8);
    buttons.setAlignment(Pos.CENTER_LEFT);
    buttons.setPadding(new Insets(6));
    for (int i = 0; i < colors.length; i += 1) {
      RadioButton rb = new RadioButton(colors[i]);
      rb.setToggleGroup(group);
      rb.setUserData(Color.web(colors[i]));
      buttons.getChildren().add(rb);
      rb.setOnAction(e -> setBtnLabel());
    }
    group.selectToggle(group.getToggles().get(0))
```

attributes: handlers have access

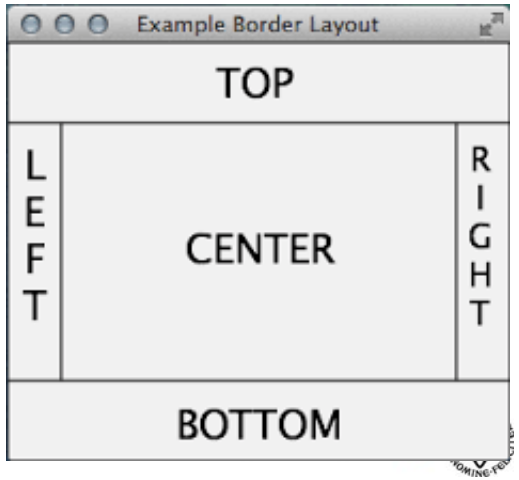
toggle group: only 1 selected

sets first radiobutton as the selected one

radio buttons 2/3

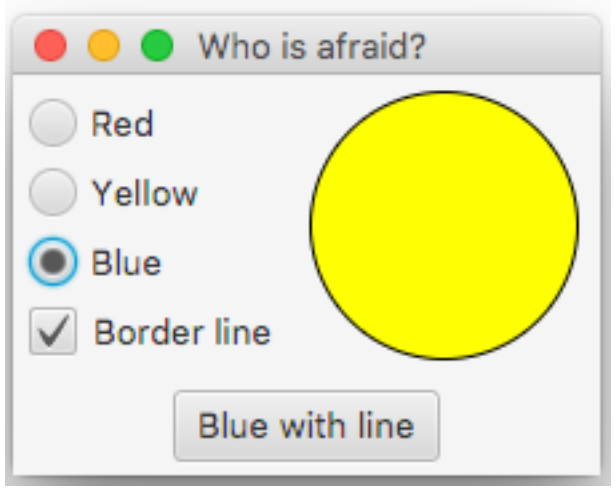
```
    cb.setOnAction(e -> setBtnLabel());
    buttons.getChildren().add(cb);
    btn.setOnAction(e -> colorCircle());
    setBtnLabel();
    colorCircle();
    BorderPane root = new BorderPane();
    root.setLeft(buttons);
    FlowPane flow = new FlowPane();
    flow.setAlignment(Pos.CENTER);
    flow.setPadding(new Insets(5));
    flow.getChildren().add(btn);
    root.setBottom(flow);
    root.setCenter(circle);
    Scene scene = new Scene(root, 220, 150);
    stage.setTitle("Who is afraid?");
    stage.setScene(scene);
    stage.show();
}
```

text indicates what will be done



radio buttons 3/3

```
private void setBtnLabel() {
  RadioButton selected = (RadioButton) group.getSelectedToggle();
  btn.setText(selected.getText() + " " +
              (cb.isSelected() ? "with line" : ""));
}
private void colorCircle() {
  Color c = (Color) group.getSelectedToggle().getUserData();
  circle.setFill(c);
  if (cb.isSelected()) {
    circle.setStroke(Color.BLACK);
  } else {
    circle.setStroke(c);
  }
}
```

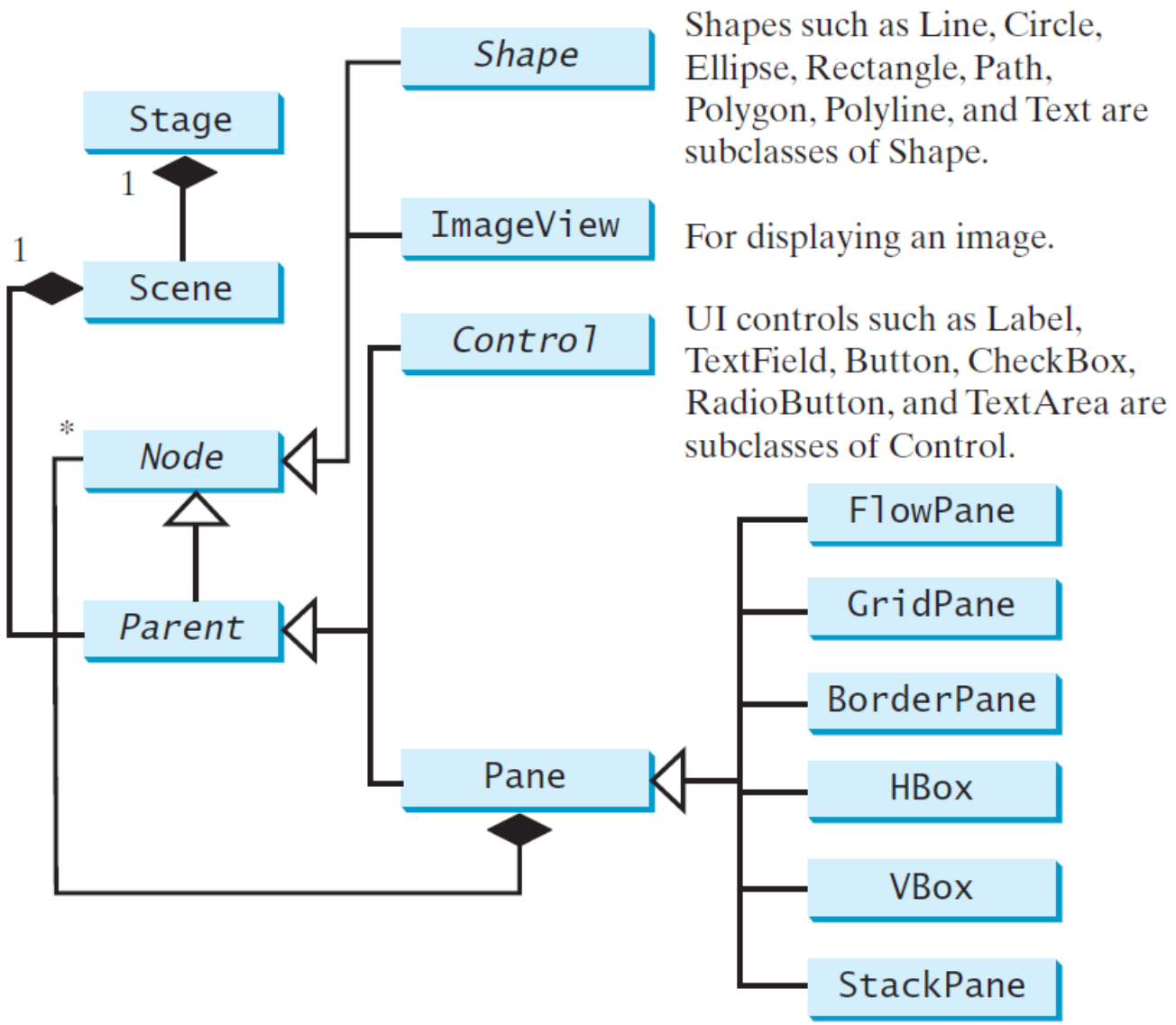


conclusion

architecture

you should know and be able to use:

- this structure
- names of important classes recognise their methods, and be able to use them
- properties
- timelines
- transformations
- MVC and Observer pattern



Lecture 10: Design Patterns