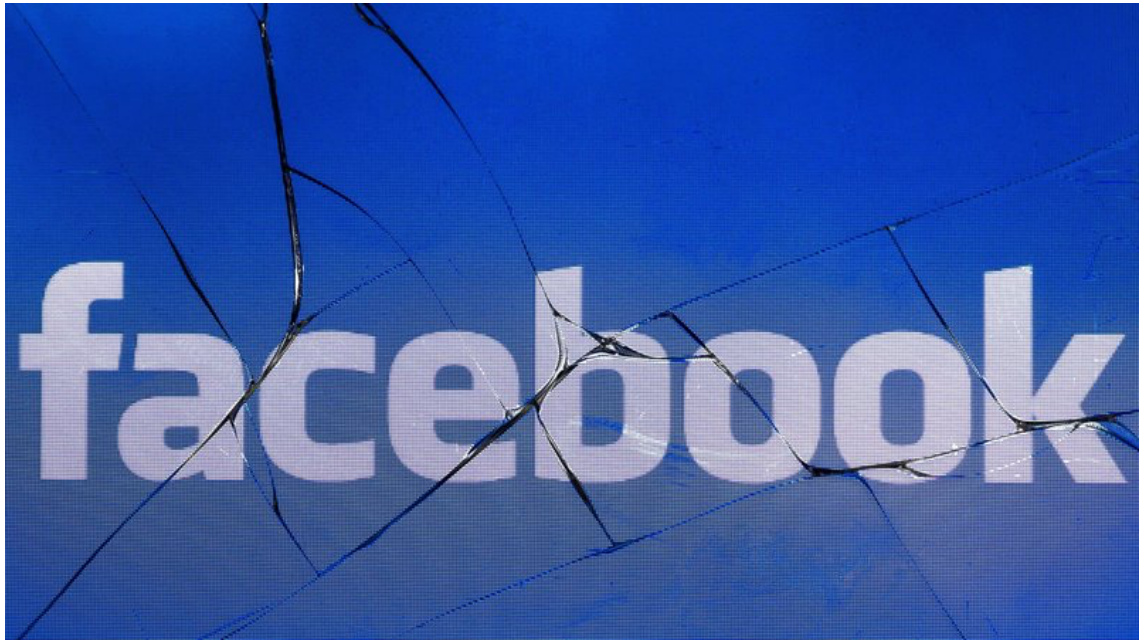
IT In the News

- 50 million Facebook accounts were affected by a security breach two weeks ago
- Attacks exploited bugs in Facebook's "View As" feature (built to give users more privacy) and a feature that allowed users to easily upload birthday videos
  - Vulnerabilities existed since July 2017
- Login tokens have been reset for those affected and vulnerabilities have been fixed
  - Still unsure of how much user data the attackers were able to see



Source:
https://www.nytimes.com/2018/09/28/technology/facebook-hack-data-breach.html
https://gizmodo.com/50-million-facebook-accounts-affected-in-massive-securi-1829394250
https://www.cnn.com/2018/10/04/tech/facebook-hack-explainer/index.html

Graphics Review: ColorChanger App (1/4)

1. Implement start in App:

- Instantiate a PaneOrganizer and a Scene, which will take in the root Pane (accessed from PaneOrganizer's getRoot() method) and width and height of the Scene
- Set the Scene, title the Stage, and show the Stage

```
public class App extends Application {

        @Override
        public void start(Stage stage) {
                PaneOrganizer organizer = new PaneOrganizer();
                Scene scene =
                new Scene(organizer.getRoot(), 100, 100);Scene
                stage.setScene(scene);
                stage.setTitle("My App!");
                stage.show();

        }

}
```

Graphics Review: ColorChanger App (2/4)

2. Set up PaneOrganizer:

- Instantiate root VBox and store in a private instance variable _root
- Create public method getRoot() that returns _root (used in App!)

```
public class PaneOrganizer {
        private VBox _root;


        public PaneOrganizer() {
                _root = new VBox();


        }

        public VBox getRoot() {
                return _root;
        }

}
```

Graphics Review: `ColorChanger` App (3/4)

3. Populate Scene Graph!

- Instantiate UI elements, like a `Button` and `Label`
- Add `btn` and `label` as children of the root

```
public class PaneOrganizer {
        private VBox _root;


        public PaneOrganizer() {
                __root = new VBox();
                Button btn = new Button(“Click Me!”);
                Label label = new Label(“Fun Label”);
                _root.getChildren().addAll(btn, label);
        }

        public VBox getRoot() {
                return _root;
        }

}
```

Graphics Review: ColorChanger App (4/4)

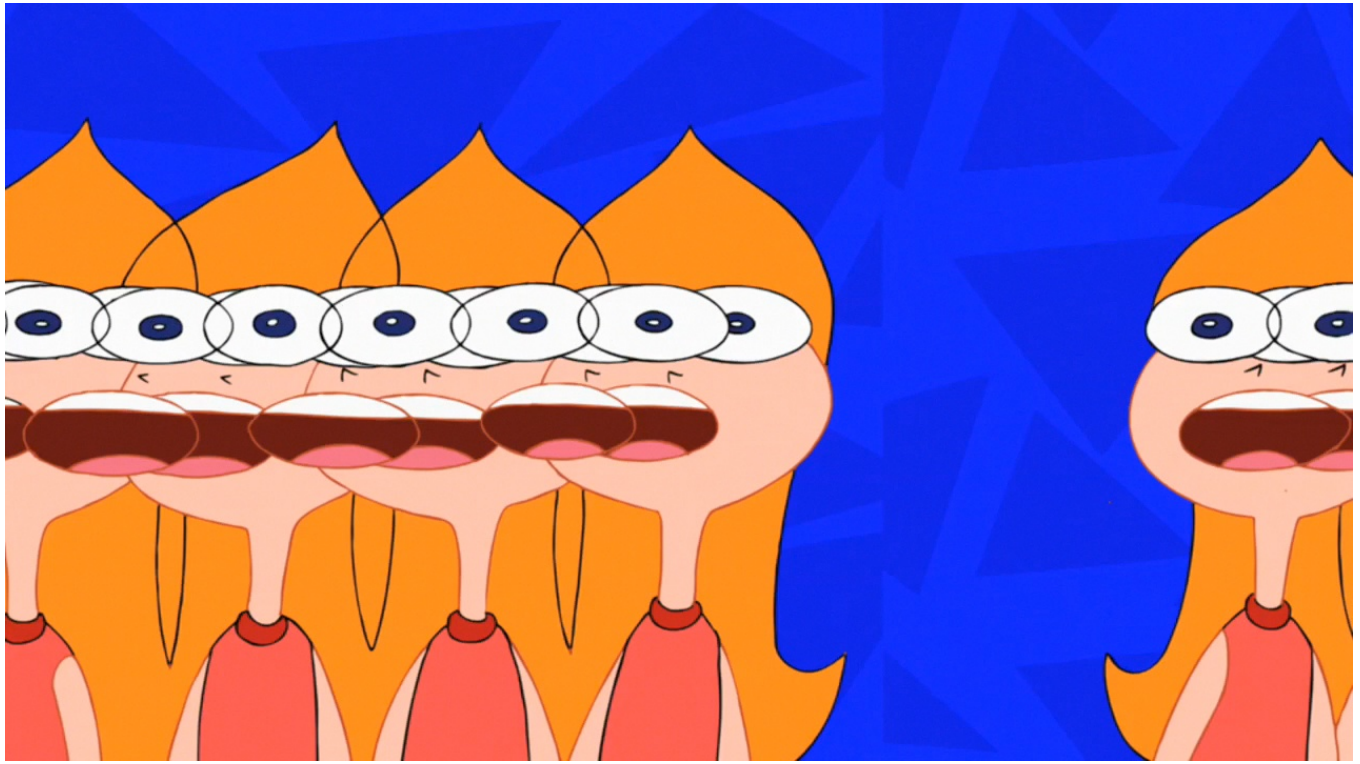
3. Define an EventHandler

- Register our btn with a ClickHandler, which is our new EventHandler class
- Create a new private class ClickHandler
- Define ClickHandler's handle() method!

```
public class PaneOrganizer {
        private VBox _root;


        public PaneOrganizer() {
                __root = new VBox();
                Button btn = new Button("Click Me!");
                Label label = new Label("Fun Label");
                _root.getChildren().addAll(btn, label);
                btn.setOnAction(new ClickHandler());
        }

        private class ClickHandler implements EventHandler<ActionEvent>() {
                public ClickHandler() { //code elided }
                public void handle(ActionEvent event) {
                        int red = (int) (Math.random()*256);
                        int green = (int) (Math.random()*256);
                        int blue = (int) (Math.random()*256);
                        Color customColor = Color.rgb(red,green,blue);
                        _label.setText(customColor);
                }
        }
}
```

Lecture 9

Graphics Part II – Understanding Animations & Shapes

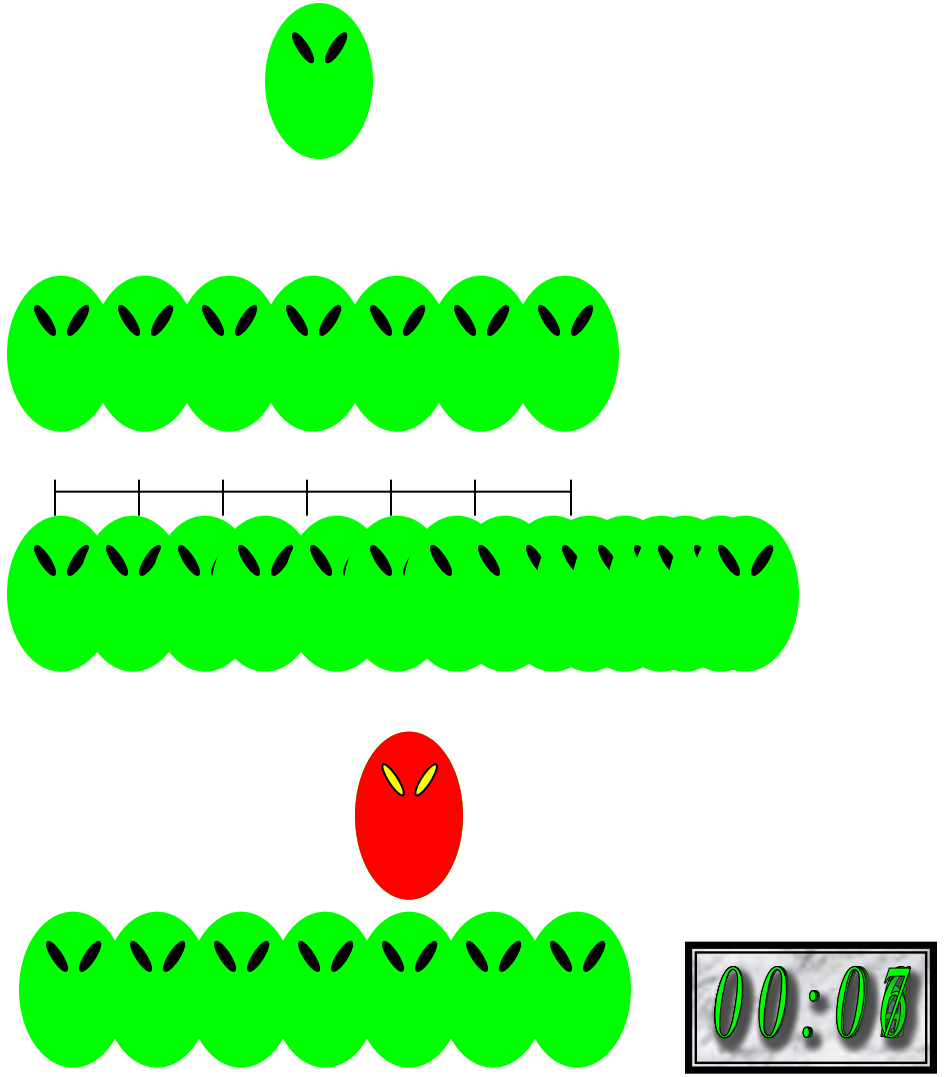


Outline

- Animation
- Layout Panes
- Java FX Shapes

Animation – Change Over Time

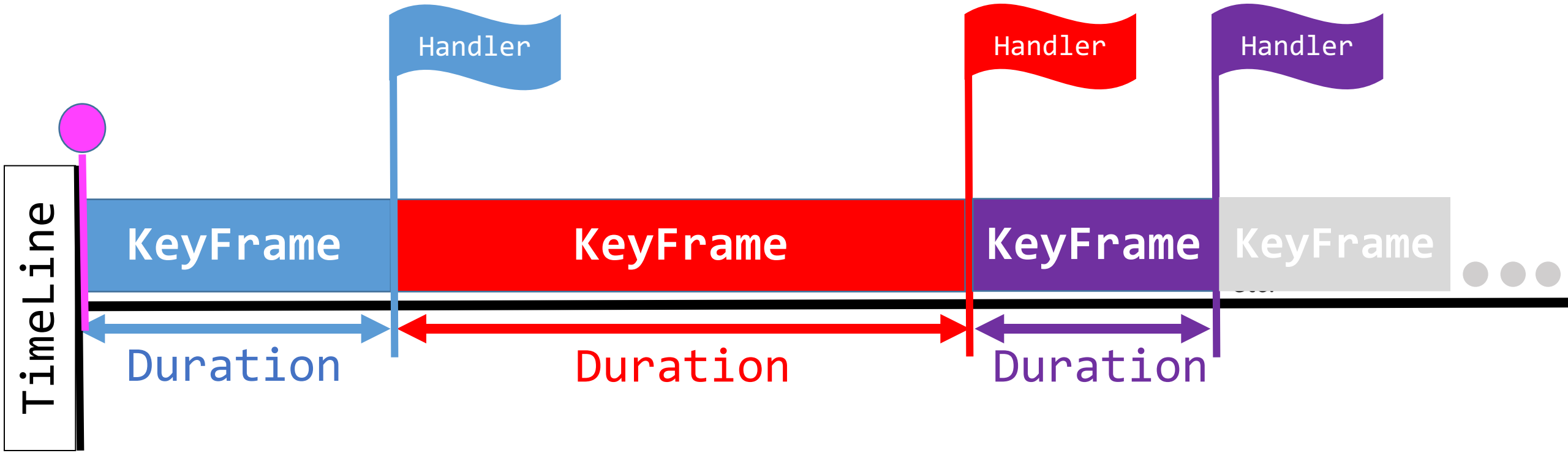
- Suppose we have an alien `Shape` we would like to **animate** (e.g. make it move across the screen)
- As in film and video animation, we can create **apparent motion** with many small changes in position
- If we move **fast enough** and in **small enough increments**, we get **smooth motion**
- Same goes for size, orientation, shape change, etc…
- How to orchestrate a sequence of incremental changes?
  - coordinate with a `Timeline` where change happens at defined instants



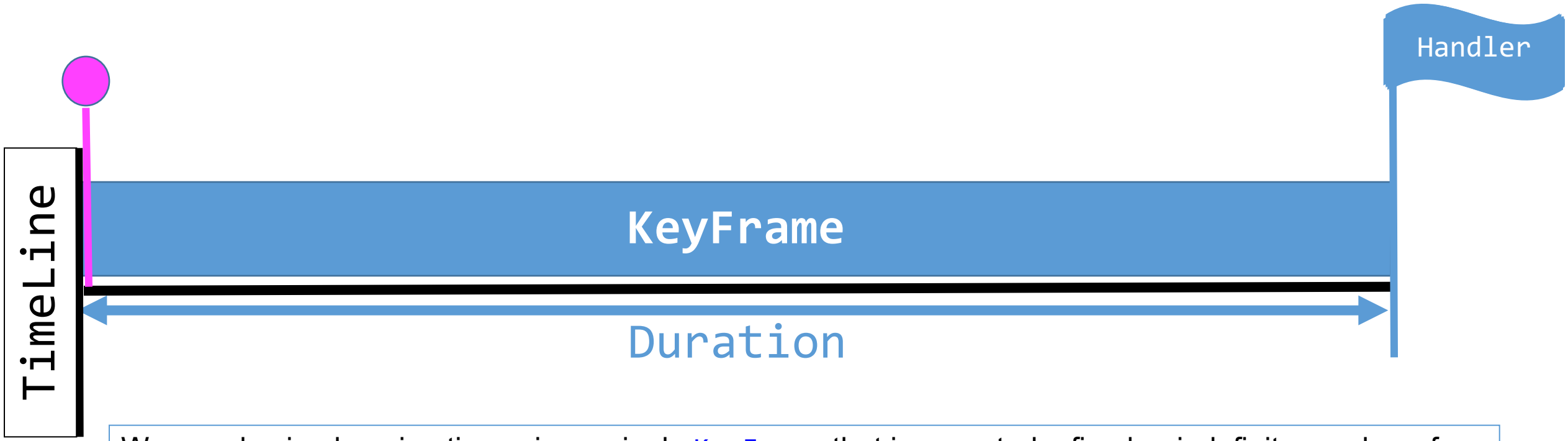
Introducing `Timelines` (1/3)

- The `Timeline` sequences one or more `KeyFrames`
  - each `KeyFrame` lasts for its entire `Duration` without making any changes
  - when the `Duration` ends, the `EventHandler` updates variables to affect the animation

Introducing Timelines (2/3)



Introducing Timelines (3/3)



We can do simple animation using a single `KeyFrame` that is repeated a fixed or indefinite number of times `EventHandler` is called, it makes incremental changes to time-varying variables (e.g., (x, y) position of a shape)

Using JavaFX `Timelines` (1/2)

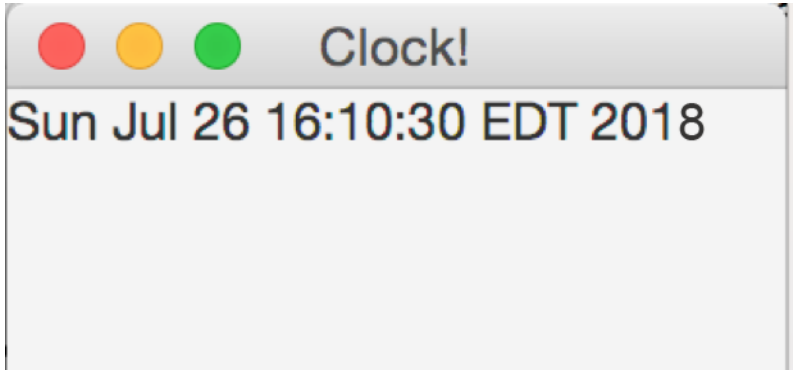
- `javafx.animation.Timeline` is used to sequence one or more `javafx.animation.KeyFrame`s, and optionally to run through them cyclically
  - each `KeyFrame` lasts for its entire `Duration` without making any changes, until its time interval ends and `EventHandler` is called to make updates
- When we instantiate a `KeyFrame`, we pass it
  - a `Duration` (e.g. `Duration.seconds(0.3)` or `Duration.millis(300)`), which defines time that each `KeyFrame` lasts
  - an `EventHandler` that defines what should occur upon completion of each `KeyFrame`
- `KeyFrame` and `Timeline` work together to control the animation, but our application's `EventHandler` is what actually causes variables to change

Using JavaFX `Timelines` (2/2)

- We then pass our new `KeyFrame` into `Timeline`
- After we instantiate our `Timeline`, we must set its `CycleCount` property
  - this defines number of cycles in `Animation`
  - we will set cycle count to `Animation.INDEFINITE`, which will let `Timeline` run forever or until we explicitly stop it
- In order for `Timeline` to work, we must then call `Timeline.play();`

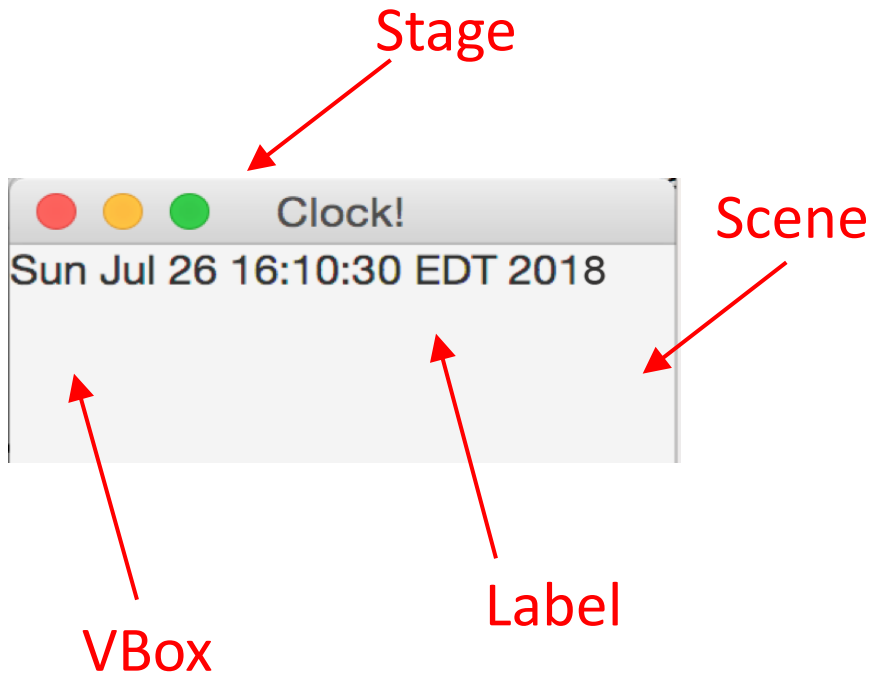
Another JavaFX App: `Clock`

- Simple example of discrete (non-smooth) animation
- Specifications: App should display current date and time, updating every second
- Useful classes:
  - `java.util.Date`
  - `javafx.util.Duration`
  - `javafx.animation.KeyFrame`
  - `javafx.animation.Timeline`



Process: `Clock`

1. **Write `App` class that extends `javafx.application.Application` and implements `start (Stage)`**
2. Write a `PaneOrganizer` class that instantiates root node and returns it in a public `getRoot()` method. Instantiate a `Label` and add it as root node's child. Factor out code for `Timeline` into its own method.
3. In our own `setupTimeline()`, instantiate a `KeyFrame` passing in a `Duration` and an instance of `TimeHandler` (defined later). Then instantiate `Timeline`, passing in our `KeyFrame`, and play `Timeline.`
4. Write private inner `TimeHandler` class that implements `EventHandler` — it should know about a `Label` and update its text on every `ActionEvent`



Clock: App class (1/3)

Note: Exactly the same process as in ColorChanger's App [Lecture 8]

1a. Instantiate a PaneOrganizer and store it in the local variable organizer

```
public class App extends Application {

    @Override
    public void start(Stage stage) {
        PaneOrganizer organizer = new PaneOrganizer();

    }

}
```

Clock: App class (2/3)

Note: Exactly the same process as in ColorChanger's App [Lecture 8]

1a. Instantiate a PaneOrganizer and store it in the local variable organizer

1b. Instantiate a Scene, passing in organizer.getRoot(), and desired width and height of Scene

```
public class App extends Application {

    @Override
    public void start(Stage stage) {
        PaneOrganizer organizer = new PaneOrganizer();
        Scene scene =
                new Scene(organizer.getRoot(), 200, 200);



    }

}
```

Clock: App class (3/3)

Note: Exactly the same process as in ColorChanger's App [Lecture 8]

1a. Instantiate a PaneOrganizer and store it in the local variable organizer

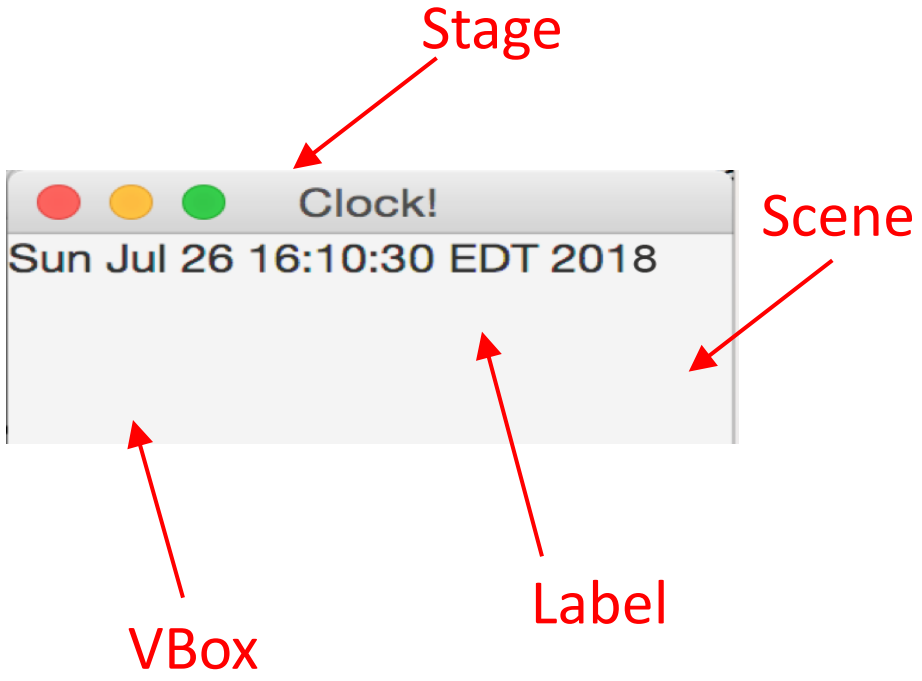
1b. Instantiate a Scene, passing in organizer.getRoot(), desired width and height of the Scene

1c. Set the Scene, set the Stage's title, and show the Stage!

```
public class App extends Application {

    @Override
    public void start(Stage stage) {
        PaneOrganizer organizer = new PaneOrganizer();
        Scene scene =
                new Scene(organizer.getRoot(), 200, 200);

        stage.setScene(scene);
        stage.setTitle("Clock!");
        stage.show();
    }

}
```

Process: Clock

1. Write App class that extends javafx.application.Application and implements start(Stage)
2. **Write a PaneOrganizer class that instantiates root node and returns it in a public getRoot() method. Instantiate a Label and add it as root node's child. Factor out code for Timeline into its own method, which we'll call setupTimeline()**
3. In our own setupTimeline(), instantiate a KeyFrame passing in a Duration and an instance of TimeHandler (defined later). Then instantiate a Timeline, passing in our KeyFrame, and play the Timeline
4. Write a private inner TimeHandler class that implements EventHandler — it should know about a Label and update its text on every ActionEvent



Clock: PaneOrganizer Class (1/3)

2a. In the PaneOrganizer class' constructor, instantiate a root VBox and set it as the return value of a public getRoot() method

```
public class PaneOrganizer{
        private VBox _root;


        public PaneOrganizer(){
            _root = new Vbox();




        }

        public VBox getRoot() {
            return _root;
        }

}
```

Clock: PaneOrganizer Class (2/3)

2a. In the PaneOrganizer class' constructor, instantiate a root VBox and set it as the return value of a public getRoot() method

2b. Instantiate a Label and add it to the list of the root node's children

```
public class PaneOrganizer{
        private VBox _root;
        private Label _label;

        public PaneOrganizer(){
            _root = new Vbox();
            _label = new Label();
            _root.getChildren().add(_label);


        }

        public VBox getRoot() {
            return _root;
        }

}
```

Clock: PaneOrganizer Class (3/3)

2a. In the PaneOrganizer class' constructor, instantiate a root VBox and set it as the return value of a public getRoot() method

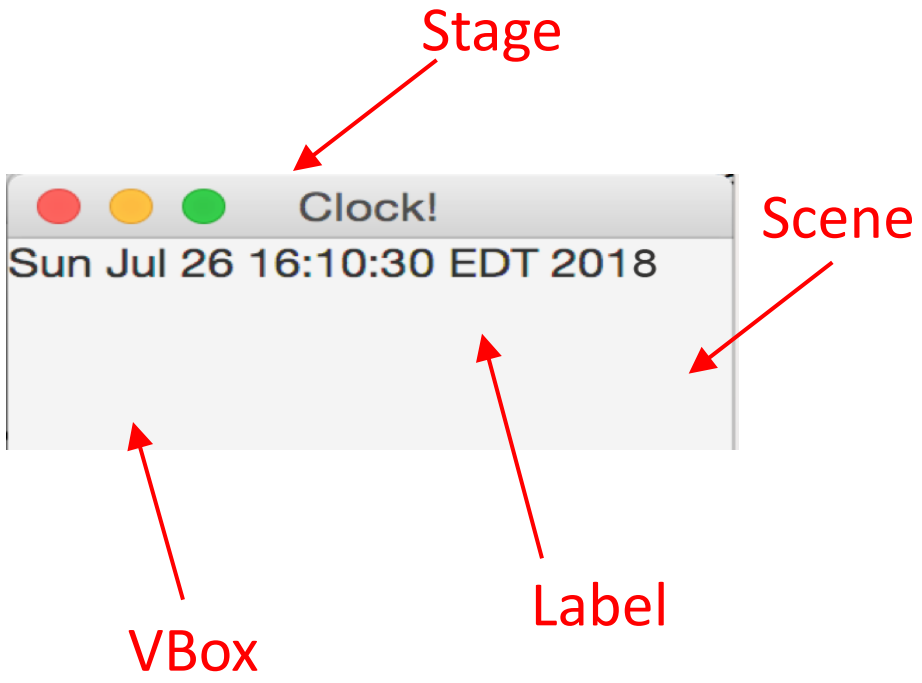
2b. Instantiate a Label and add it to the list of the root node's children

2c. Call setupTimeline(); this is another example of delegation to a specialized "helper method" which we'll define next !

```
public class PaneOrganizer{
        private VBox _root;
        private Label _label;

        public PaneOrganizer(){
            _root = new Vbox();
            _label = new Label();
            _root.getChildren().add(_label);

            this.setupTimeline();
        }

        public VBox getRoot() {
            return _root;
        }

}
```

Process: `Clock`

1. Write an `App` class that extends `javafx.application.Application` and implements `start(Stage)`
2. Write a `PaneOrganizer` class that instantiates the root node and returns it in a public `getRoot()` method. Instantiate a `Label` and add it as the root node's child. Factor out code for `Timeline` into its own method
3. **In `setupTimeline()`, instantiate a `KeyFrame`, passing in a `Duration` and an instance of `TimeHandler` (defined later). Then instantiate a `Timeline`, passing in our `KeyFrame`, and play the `Timeline`.**
4. Write a private inner `TimeHandler` class that implements `EventHandler` — it should know about a `Label` and update its text on every `ActionEvent`



Clock: PaneOrganizer class- setupTimeline() (1/4)

Within `setupTimeline()`:

3a. Instantiate a `KeyFrame`, which takes two parameters

- want to update text of `_label` each second — therefore make `Duration` of the `KeyFrame` 1 second
- for the `EventHandler` parameter pass an instance of our `TimeHandler` class, to be created later

```
public class PaneOrganizer{
    //other code elided

    public void setupTimeline(){
        KeyFrame kf = new KeyFrame(
                  Duration.seconds(1), //how long
                  new TimeHandler()); //event handle




    }

}
```

Note: JavaFX automatically calls `TimeHandler`'s `handle()` method at end of `KeyFrame`, which in this case changes the label text, and then lets next 1 second cycle of `KeyFrame` start

Clock: PaneOrganizer class- setupTimeline() (2/4)

Within `setupTimeline()`:

3a. Instantiate a `KeyFrame`

3b. Instantiate a `Timeline`, passing in our new `KeyFrame`

```
public class PaneOrganizer{
    //other code elided

    public void setupTimeline(){
        KeyFrame kf = new KeyFrame(
                  Duration.seconds(1),
                  new TimeHandler());

        Timeline timeline = new Timeline(kf);



    }

}
```

Clock: PaneOrganizer class - setupTimeline() (3/4)

Within `setupTimeline()`:

3a. Instantiate a `KeyFrame`

3b. Instantiate a `Timeline`, passing in our new `KeyFrame`

3c. Set `CycleCount` to `INDEFINITE`

```
public class PaneOrganizer{
    //other code elided

    public void setupTimeline(){
        KeyFrame kf = new KeyFrame(
                Duration.seconds(1),
                new TimeHandler());

        Timeline timeline = new Timeline(kf);

        timeline.setCycleCount(
                Animation.INDEFINITE);

    }

}
```

Clock: PaneOrganizer class- setupTimeline() (4/4)

Within `setupTimeline()`:

3a. Instantiate a `KeyFrame`

3b. Instantiate a `Timeline`, passing in our new `KeyFrame`

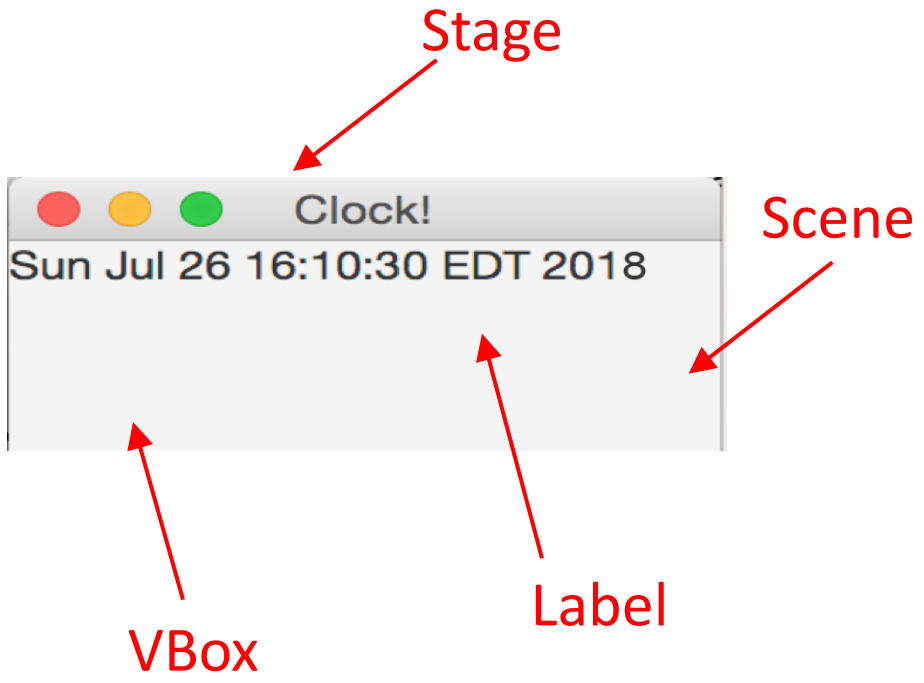
3c. Set `CycleCount` to `INDEFINITE`

3d. Play, i.e. start `Timeline`

```
public class PaneOrganizer{
    //other code elided

    public void setupTimeline(){
        KeyFrame kf = new KeyFrame(
                  Duration.seconds(1),
                  new TimeHandler());

        Timeline timeline = new Timeline(kf);

        timeline.setCycleCount(
                  Animation.INDEFINITE);
        timeline.play();
    }

}
```

Process: `Clock`

1. Write an `App` class that extends `javafx.application.Application` and implements `start(Stage)`
2. Write a `PaneOrganizer` class that instantiates the root `Node` and returns it in public `getRoot()` method. Instantiate a `Label` and add it as root `node`'s child. Factor out code for `Timeline` into its own method.
3. In `setupTimeline()`, instantiate a `KeyFrame` passing in a `Duration` and an instance of `TimeHandler` (defined later). Then instantiate a `Timeline`, passing in our `KeyFrame`, and play the `Timeline`.
4. **Write a private inner `TimeHandler` class that implements `EventHandler` — it should know about a `Label` and update its text on every `ActionEvent`**



Clock: TimeHandler Private Inner Class (1/3)

4a. The last step is to create our TimeHandler and implement handle(), specifying what should occur **at the end** of each KeyFrame – called automatically by JFX

```
public class PaneOrganizer{
        //other code elided

        private class TimeHandler implements
                EventHandler<ActionEvent>{

            public void handle(ActionEvent event){




            }

        } //end of private TimeHandler class

} //end of PaneOrganizer class
```

Clock: TimeHandler Private Inner Class (2/3)

4a. The last step is to create our TimeHandler and implement handle(), specifying what to occur **at the end** of each KeyFrame – called automatically by JFX

4b. java.util.Date represents a specific instant in time. Date is a representation of the time, to the nearest millisecond, at the moment the Date is instantiated

```
public class PaneOrganizer{
        //other code elided

        private class TimeHandler implements
                EventHandler<ActionEvent>{

            public void handle(ActionEvent event){
                Date now = new Date();



            }

        } //end of private TimeHandler class

} //end of PaneOrganizer class
```

Clock: TimeHandler Private Inner Class (3/3)

4a. The last step is to create our TimeHandler and implement handle(), specifying what to occur **at the end** of each KeyFrame – called automatically by JFX

4b. java.util.Date represents a specific instant in time. Date is a representation of the time, to the nearest millisecond, at the moment the Date is instantiated

4c. Because our Timeline has a Duration of 1 second, each second a new Date will be generated, converted to a String, and set as the _label's text. This will appropriately update _label with correct time every second!

```
public class PaneOrganizer{
        //other code elided

        private class TimeHandler implements
                EventHandler<ActionEvent>{

            public void handle(ActionEvent event){
                Date now = new Date();
                //toString converts the Date into a
                //String with year, day, time etc.

                //_label instantiated in
                //constructor of PO
                _label.setText(now.toString());
            }

        } //end of private TimeHandler class

} //end of PaneOrganizer class
```

The Whole App: Clock

```
//App class imports
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.application.*;
// package includes Pane class and its subclasses
import javafx.scene.layout.*;
//package includes Label, Button classes
import javafx.scene.control.*;
//package includes ActionEvent, EventHandler classes
import javafx.event.*;
import javafx.util.Duration;
import javafx.animation.Animation;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import java.util.Date;
public class App extends Application {

        @Override
        public void start(Stage stage) {
        PaneOrganizer organizer = new PaneOrganizer();
        Scene scene = new Scene(organizer.getRoot(), 200, 200);

        stage.setScene(scene);
        stage.setTitle("Clock");
        stage.show();
        }

}
```

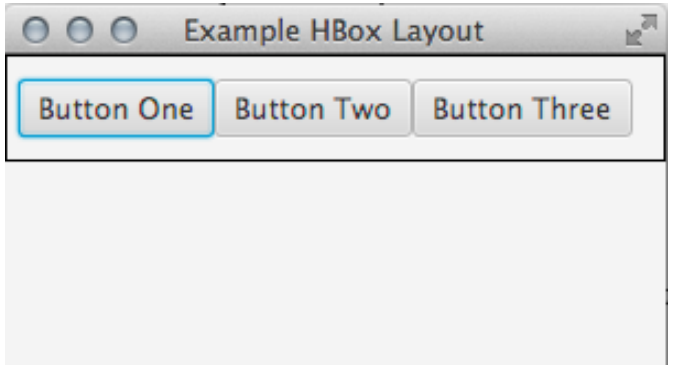
```
public class PaneOrganizer{
        private VBox _root;
        private Label _label;

        public PaneOrganizer(){
            _root = new VBox();
            _label = new Label();
            _root.getChildren().add(_label);
            this.setupTimeline();
        }

        public VBox getRoot() {
            return _root;
        }

        public void setupTimeline(){
            KeyFrame kf = new KeyFrame(Duration.seconds(1),
                  new TimeHandler());
            Timeline timeline = new Timeline(kf);
            timeline.setCycleCount(Animation.INDEFINITE);
            timeline.play();
        }

        private class TimeHandler
        implements EventHandler<ActionEvent>{
            public void handle(ActionEvent event){
                Date now = new Date();
                _label.setText(now.toString());
            }
        }

}
```

Layout Panes

- Until now, we have been adding all our GUI components to a VBox
  - VBoxes lay everything out in one vertical column
- What if we want to make some more interesting GUIs?
- Use different types of layout panes!
  - VBox is just one of many JavaFX panes—there are many more options
  - we will introduce a few, but check out our documentation or JavaDocs for a complete list

HBox

- Similar to `Vbox`, but lays everything out in a horizontal row (hence the name)
- Example:

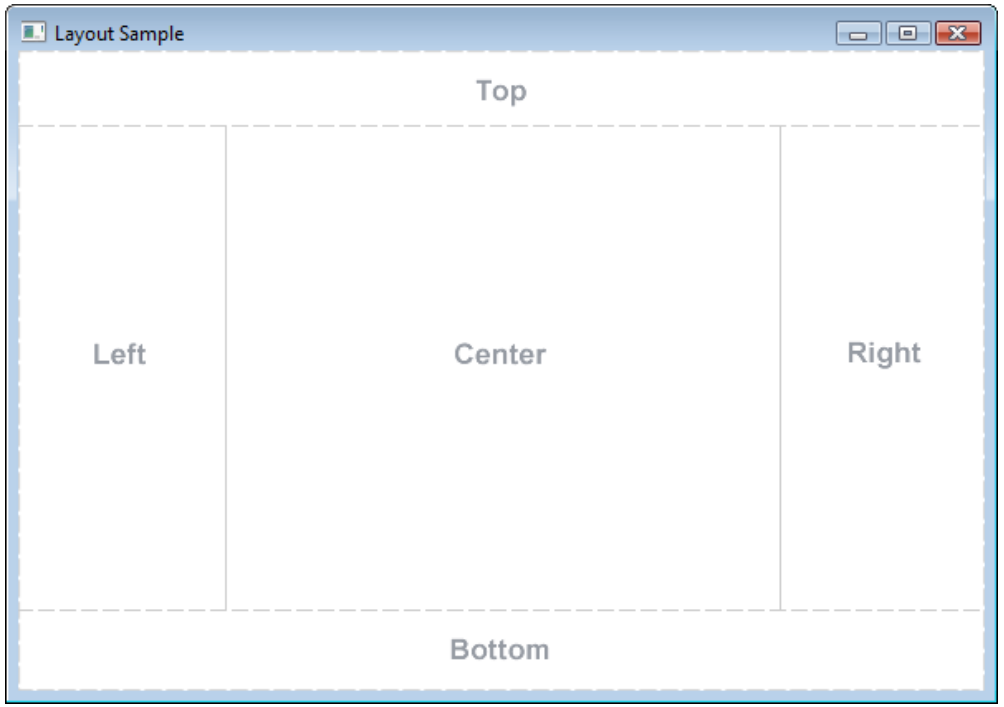



```
// code for setting the scene elided
HBox buttonBox = new HBox();
Button b1 = new Button("Button One");
Button b2 = new Button("Button Two");
Button b3 = new Button("Button Three");
buttonBox.getChildren().addAll(b1, b2, b3);
```

- Like `VBox`, we can set the amount of horizontal spacing between each child in the `HBox` using the `setSpacing(double)` method

`BorderPane` (1/2)

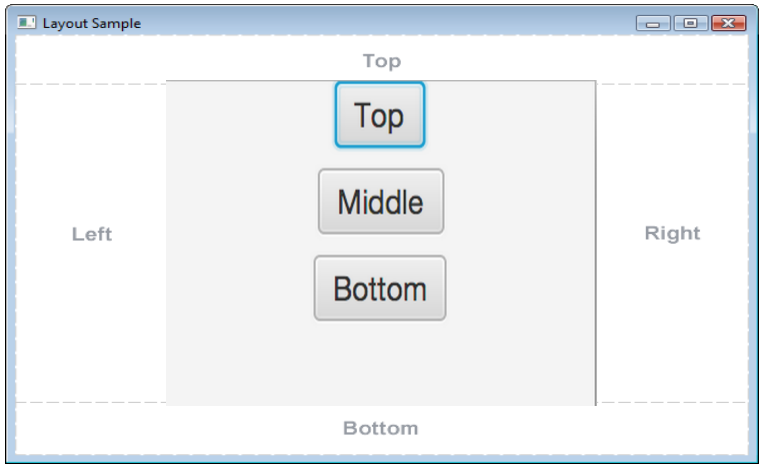
- `BorderPane` lays out children in top, left, bottom, right and center positions
- To add things visually, use `setLeft(Node)`, `setCenter(Node)`, etc.
  - this includes an implicit call to `getChildren().add(…)`
- Use any type of `Node`—`Panes` (with their own children), `Buttons`, `Labels`, etc.!



BorderPane (2/2)

- Remember our VBox example from earlier?

```
VBox buttonBox = new VBox();
Button b1 = new Button("Top");
Button b2 = new Button("Middle");
Button b3 = new Button("Bottom");
buttonBox.getChildren.addAll(b1,b2,b3);
buttonBox.setSpacing(8);
buttonBox.setAlignment(Pos.TOP_CENTER);
```



- We can make our VBox the center of this BorderPane

```
BorderPane container = new BorderPane();
container.setCenter(buttonBox);
```

Note: we didn't have to call `container.getChildren.add(buttonBox)`, as this call is done implicitly in the `setCenter()` method!

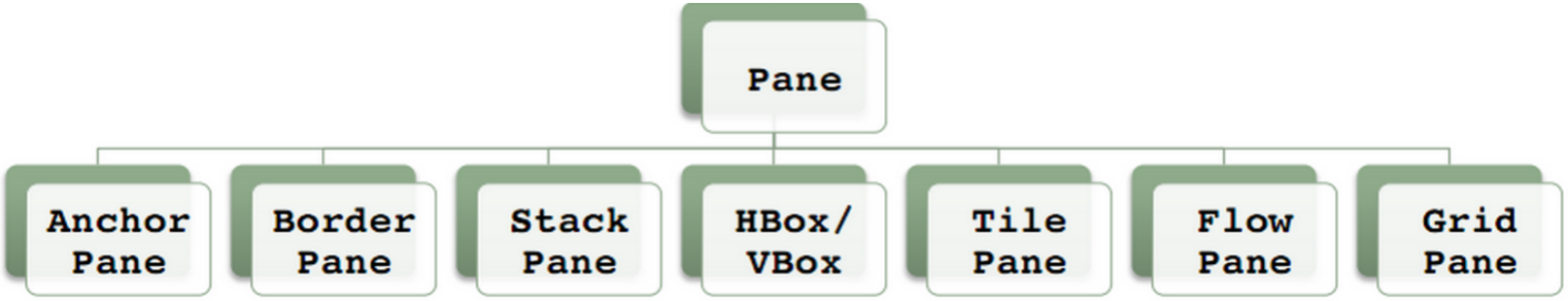
- No need to use all regions—could just use a few of them
- Unused regions are "compressed", e.g. could have a two-region (left/right) layout without a center

Absolute Positioning

- Until now, all layout panes we have seen have performed layout management for us
  - what if we want to position our GUI components freely ourselves?
- Need to set component's location to exact *pixel location* on screen
  - called *absolute positioning*
- When would you use this?
  - to position shapes—stay tuned!

Pane

- `Pane` allows you to lay things out completely freely, like on an art canvas
- It is a concrete superclass to all more specialized layout panes seen earlier that do automatic positioning
  - can call methods on its children (panes, buttons, shapes, etc.) to set location within pane
    - for example: use `setX(double)` and `setY(double)` to position a `Rectangle`
  - `Pane` performs no layout management, so coordinates you set determine where things appear on the screen

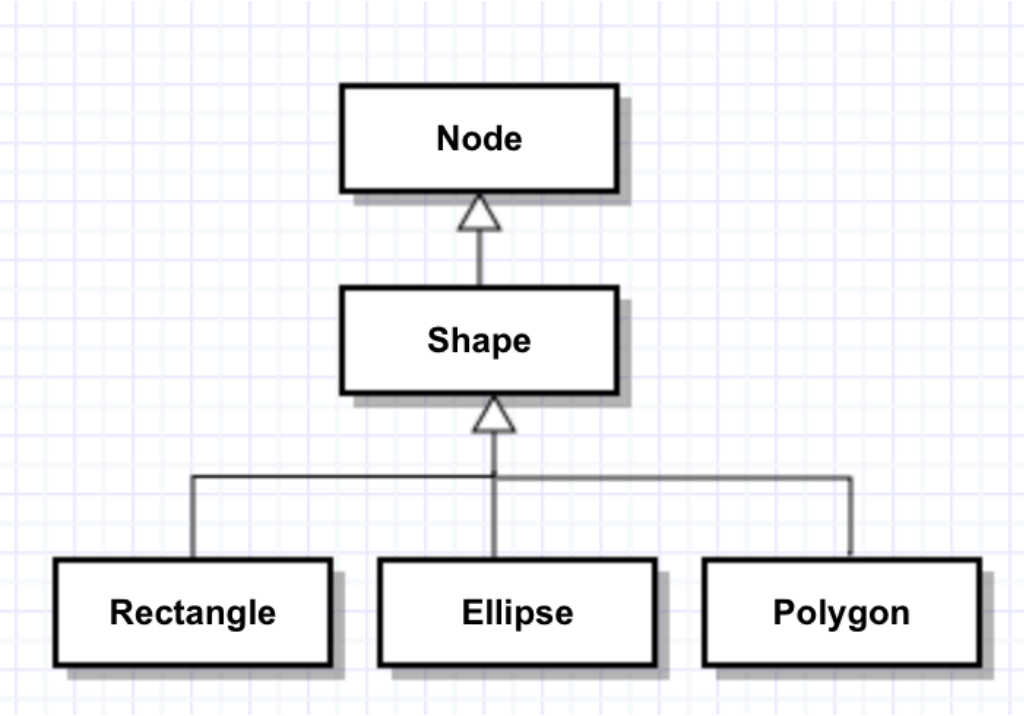


Creating Custom Graphics

- We've now introduced you to using JavaFX's native UI elements
  - ex: `Label` and `Button`
- Lots of handy widgets for making your own graphical applications!
- What if you want to create your own custom graphics?
- This lecture: build your own graphics using the `javafx.scene.shape` package!

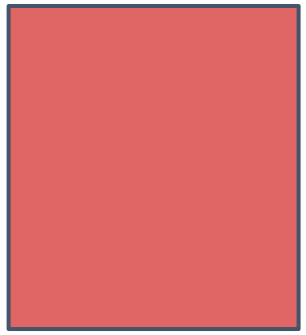
`javafx.scene.shape` Package

- JavaFX provides built-in classes to represent 2D shapes, such as rectangles, ellipses, polygons, etc.
- All these classes inherit from abstract class `Shape`, which inherits from `Node`
  - methods relating to rotation and visibility are defined in `Node`
  - methods relating to color and border are defined in `Shape`
  - other methods are implemented in the individual classes of `Ellipse`, `Rectangle`, etc.

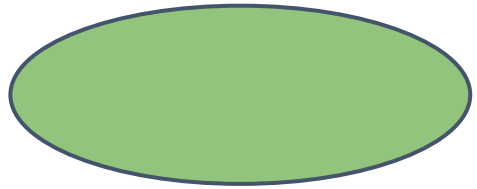


Shape Constructors

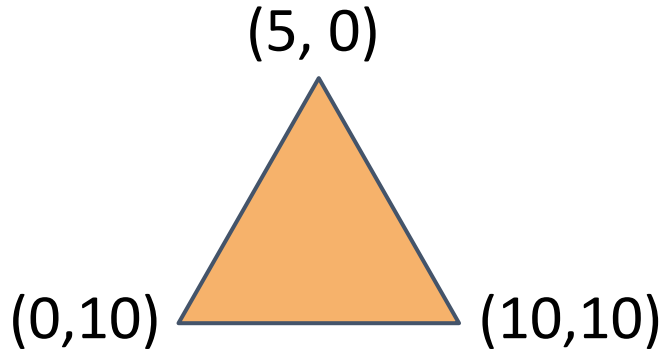
Default position for Shape with this constructor would be (0,0)



- `Rectangle(double width, double height)`
- `Ellipse(double radiusX, double radiusY)`
- `Polygon(double … points)`
  - the "…" in the signature means that you can pass in as many points as you would like to the constructor
  - pass in `Points` (even number of x and y coordinates) and `Polygon` will connect them for you
  - passing points will define and position the shape of `Polygon`- this is not always the case with other `Shape`s (like `Rectangle` or `Ellipse`)
  - Example: `new Polygon(0,10,10,10,5,0)`

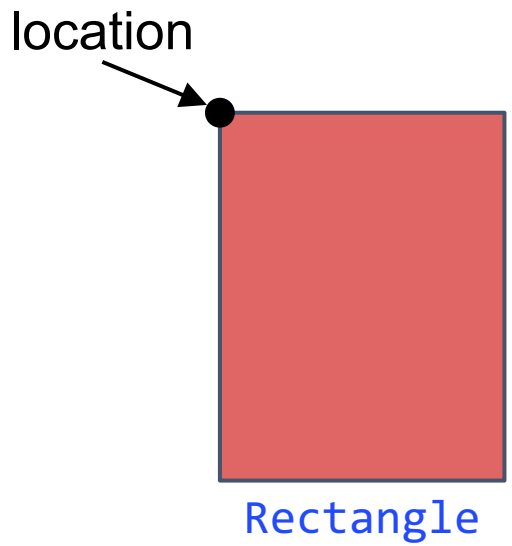


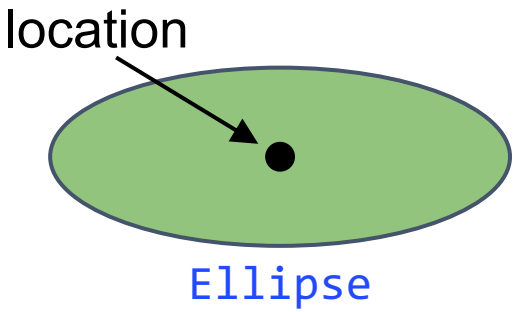
- Each of these `Shape` subclasses have multiple overloaded constructors (see Math and Making Decisions, slide 51) — check out the JavaFX documentation for more options!
  - for example, if you wanted to instantiate a `Rectangle` with a given position and size: `Rectangle(double x, double y, double width, double height)`

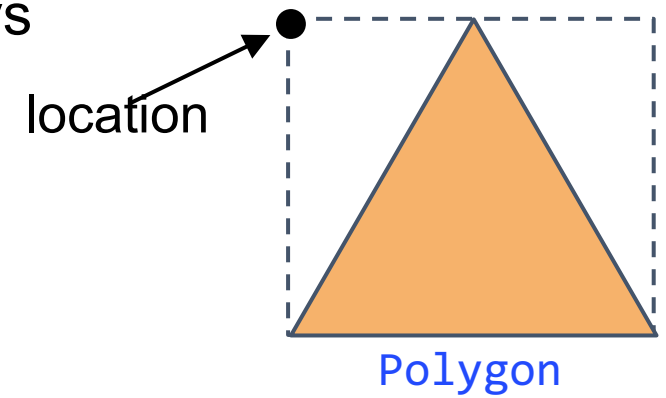


Shapes: Setting Location

- JavaFX `Shape`s have different behaviors (methods) for setting their location within their parent's coordinate system
  - `Rectangle`: use `setX(double)` and `setY(double)`
  - `Ellipse`: use `setCenterX(double)` and `setCenterY(double)`
  - `Polygon`: use `setLayoutX(double)` and `setLayoutY(double)`
- JavaFX has *many* different ways to set location
  - from our experience, these are the most straightforward ways
  - if you choose to use other methods, be sure you fully understand them or you may get strange bugs!
  - check out our JavaFX documentation and the Javadocs for more detailed explanations!

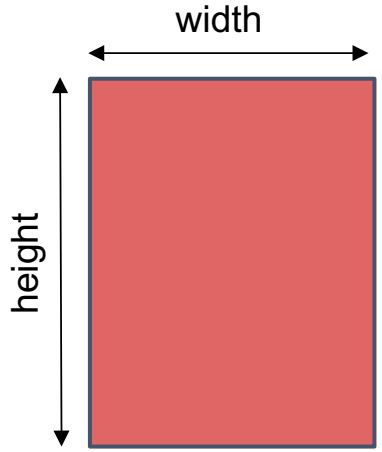


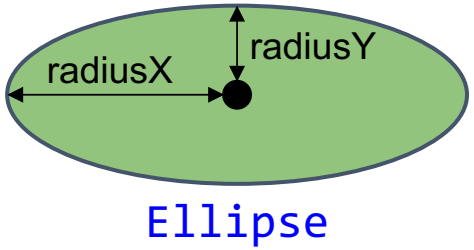


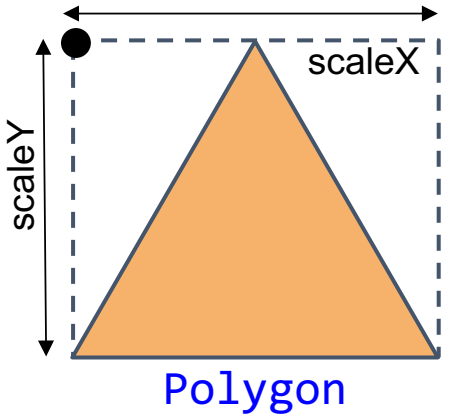


Shapes: Setting Size

- JavaFX Shapes also have different behaviors (methods) for altering their size
  - Rectangle: use setWidth(double) and setHeight(double)
  - Ellipse: use setRadiusX(double) and setRadiusY(double)
  - Polygon: use setScaleX(double) and setScaleY(double)
    - multiplies the original size in the X or Y dimension by the **scale factor**
- Again, this is not the only way to set size for Shapes but it is relatively painless
  - reminder: JavaFX documentation and Javadocs!







Accessors and Mutators of all Shapes

`final` = can't override method

- Rotation:
  - `public final void setRotate(double rotateAngle);`
  - `public final double getRotate();`
- Visibility:
  - `public final void setVisible(boolean visible);`
  - `public final boolean getVisible();`
- Color:
  - `public final void setStroke(Paint value);`
  - `public final Paint getStroke();`
  - `public final void setFill(Paint value);`
  - `public final Paint getFill();`
- Border:
  - `public final void setStrokeWidth(double val);`
  - `public final double getStrokeWidth();`

Rotation is about the center of the `Shape`'s "bounding box"; i.e., the smallest rectangle that contains the entire shape. To have a `Shape` rotate about an arbitrary center of rotation, add a `Rotate` instance to the `Shape` and set a new center of rotation (see Javadocs)

The stroke is the border that outlines the `Shape`, while the fill is the color of the interior of the `Shape`

Generally, uses a `Color`, which inherits from `Paint`. Use predefined color constants `Color.WHITE`, `Color.BLUE`, `Color.AQUA`, etc., or define your own new color by using the following syntax:

```
Paint color = Color.color(0.5, 0.5, 0.5);
```

OR:

```
Paint color = Color.rgb(100, 150, 200);
```

Announcements

- FruitNinja deadlines:
  - Early: Friday, 10/5 at 11:59pm
  - On-time: Sunday, 10/7 at 11:59pm
  - Late: Tuesday, 10/9 at 11:59pm
  - Note: **No hours** on Indigenous People's Day! Plan accordingly
- Mentorship pairings have been made! If you signed up, your mentor will reach out soon
  - If you'd like a mentor, reach out to the HTAs!