Lecture 10

Graphics Part II – Animations & Shapes

Outline

- EventHandlers
- Lamda Expressions
- Animation
- Layout Panes
- Java FX Shapes

EventHandlers (1/3)

- Button click causes JavaFX to generate a javafx.event.ActionEvent
  - ActionEvent is only one of many JavaFX EventTypes that are subclasses of Event class
- Classes that implement EventHandler interface can polymorphically handle any subclass of Event
  - when a class implements EventHandler interface, it must specify what type of Event it should know how to handle
  - how do we do this?

EventHandlers (2/3)

- `EventHandler` interface declared as:

```
public interface EventHandler<T extends Event>…
```

  - the code inside literal < > is known as a “generic parameter” – this is magic for now
  - lets you specialize the interface method declarations to handle one specific specialized subclass of `Event`
  - forces *you* to replace what is inside the literal < > with some subclass of `Event`, such as `ActionEvent`, whenever *you* write a class that implements `EventHandler` interface

EventHandlers (3/3)

- `EventHandler` interface only has one method, the `handle` method
- Parameter of `handle` will match the generic parameter of `EventHandler` type
  - in this case `ActionEvent` since `Button`s generate `ActionEvent`s
  - JavaFX generates the specific event for you and passes it as an argument to your `handle` method
  - Note we don't actually use the data contained in an `ActionEvent` parameter for button click handlers, but for `MouseEvent`s and `KeyEvent`s, you will need to use the event parameter (next lecture!)

Method Summary

All Methods | **Instance Methods** | **Abstract Methods**

Modifier and Type	Method and Description
`void`	`handle(T event)` Invoked when a specific event of the type for which this handler is registered happens.

Registering an `EventHandler` (1/2)

- How do we let a `Button` know which `EventHandler` to execute when it's clicked?
- We must **register** the `EventHandler` with the `Button` via the `Button`'s `setOnAction` method so that JavaFX can store the association with the `EventHandler` and call it when the `Button` is clicked
  - note the "generic parameter" `<ActionEvent>` since button clicks generate `ActionEvent`s

setOnAction

```
public final void setOnAction(EventHandler<ActionEvent> value)
```

Sets the value of the property onAction.

Property description:

The button's action, which is invoked whenever the button is fired. This may be due to the user clicking on the button with the mouse, or by a touch event, or by a key press, or if the developer programmatically invokes the `fire()` method.

Registering an EventHandler (2/2)

1. Write custom EventHandler class (MyClickHandler), implementing handle with previous code to generate Color
   - must create an **association** with the Label so the handler knows which Label to change
2. In PaneOrganizer, register the EventHandler with the Button, using setOnAction method
3. When Button is clicked, handle method in MyClickHandler is passed an ActionEvent by JavaFX and is then executed

```
public class MyClickHandler implements EventHandler<ActionEvent> {
    private Label label;
    public MyClickHandler(Label myLabel) {
        this.label = myLabel;
    }

    @Override
    public void handle(ActionEvent e) {
       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);
       this.label.setTextFill(customColor);
    }
}

public class PaneOrganizer {

    public PaneOrganizer() {
        // previous code elided
        Label label = new Label("CS15 Rocks");
        Button btn = new Button("Random Color");
        btn.setOnAction(new MyClickHandler(label));
    }
}
```

Outline

- EventHandlers
- Lamda Expressions
- Animation
- Layout Panes
- Java FX Shapes

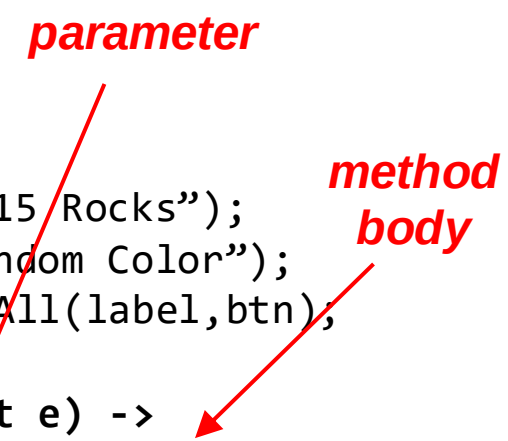
Lambda Expressions (1/3)

- Creating a separate class `MyClickHandler` is not the most efficient solution
  - more complex `EventHandler`s may have tons of associations with other nodes, all to implement one `handle` method
- Since `EventHandler` interface only has one method, we can use special syntax called a **lambda expression** instead of defining a separate class for implementation of `handle`

Lambda Expressions (2/3)

- **Lambda expressions** have different syntax with same semantics as typical method
  - first **parameter list**
  - followed by **->**
  - then an arbitrarily complex **method body** in curly braces
    - in CS15, lambda expression body will be one line calling another method, typically written yourself in the same class; in this case `changeLabelColor`
    - can omit curly braces when method body is one line

```
public class PaneOrganizer {
    private VBox root;

    public PaneOrganizer() {
        this.root = new VBox();
        Label label = new Label("CS15 Rocks");
        Button btn = new Button("Random Color");
        this.root.getChildren().addAll(label,btn);
        this.root.setSpacing(8);
        btn.setOnAction((ActionEvent e) ->
                        this.changeLabelColor(label));
    }

    public void changeLabelColor(Label myLabel) {
        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);
        myLabel.setTextFill(customColor);
    }
}
```



Lambda Expressions (3/3)

- Lambda expression shares **scope** with its enclosing method
  - can access `myLabel` or `btn` without setting up a class association
- Lambda expression body is then stored by JavaFX to be called once the button is clicked

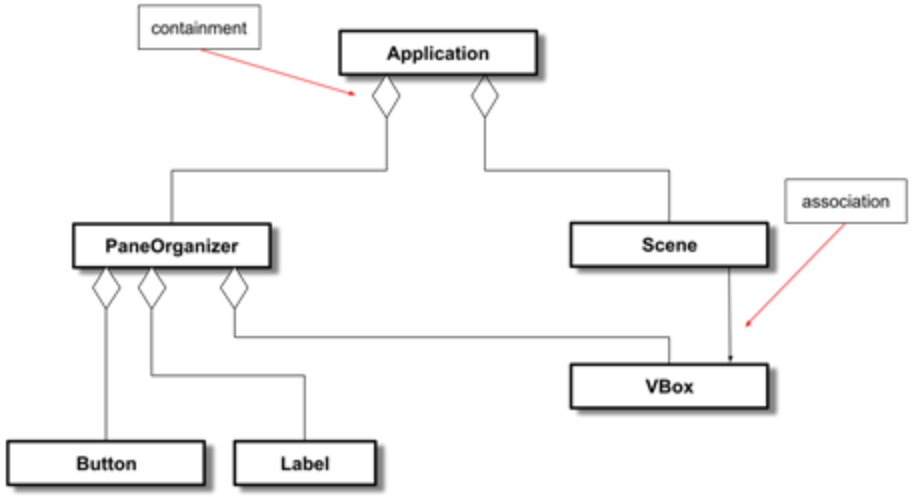
```
public class PaneOrganizer {
    private VBox root;

    public PaneOrganizer() {
        this.root = new VBox();
        Label label = new Label("CS15 Rocks");
        Button btn = new Button("Random Color");
        this.root.getChildren().addAll(label,btn);
        this.root.setSpacing(8);
        btn.setOnAction((ActionEvent e) ->
                        this.changeLabelColor(label));
    }

    public void changeLabelColor(Label myLabel) {
        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);
        myLabel.setTextFill(customColor);
    }
}
```

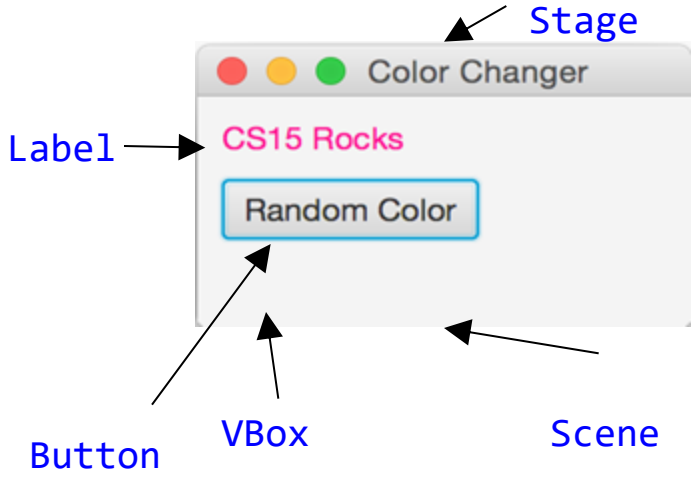
The Whole App: ColorChanger

```
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.application.Application;

public class App extends Application {

    @Override
    public void start(Stage stage) {
        PaneOrganizer organizer = new PaneOrganizer();
        Scene scene = new Scene(organizer.getRoot(),180,80);
        stage.setScene(scene);
        stage.setTitle("Color Changer");
        stage.show();
    }
}
```

```
import javafx.scene.layout.VBox;
import javafx.scene.control.Label;
import javafx.scene.control.Button;
import javafx.event.ActionEvent;
import javafx.scene.paint.Color;

public class PaneOrganizer {
    private VBox root;

    public PaneOrganizer() {
        this.root = new VBox();
        Label label = new Label(“CS15 Rocks”);
        Button btn = new Button(“Random Color”);
        this.root.getChildren().addAll(label,btn);
        this.root.setSpacing(8);
        btn.setOnAction((ActionEvent event) ->
                                    this.changeLabelColor(label));
    }

    public VBox getRoot() {
        return this.root;
    }

    private void changeLabelColor(Label myLabel) {
        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);
        myLabel.setTextFill(customColor);
    }
}
```

Note: Logical vs. Graphical Containment/Scene Graph





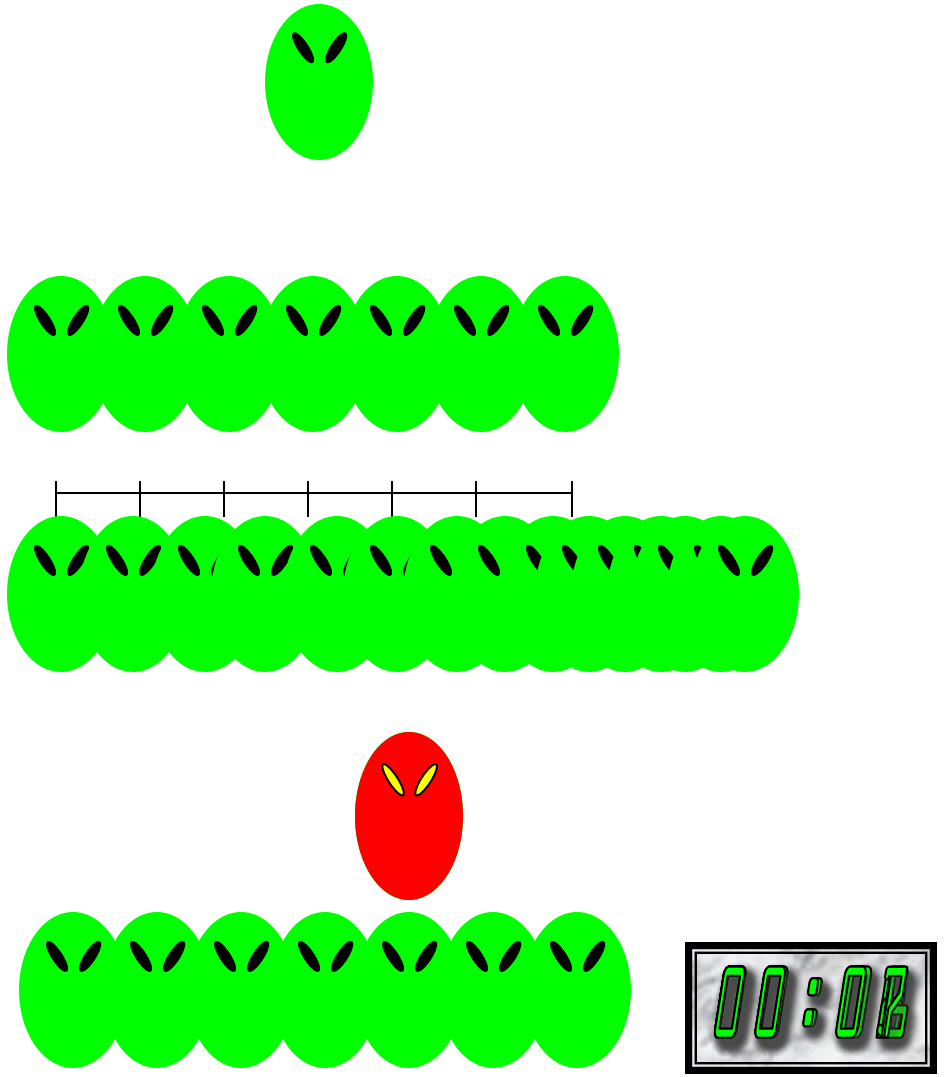
- *Graphically*, `VBox` is a `pane` contained within `Scene`, but *logically*, `VBox` is contained within `PaneOrganizer`
- *Graphically*, `Button` and `Label` are contained within `VBox`, but *logically*, `Button` and `Label` are contained within `PaneOrganizer`, which has no graphical appearance
- *Logical* containment is based on where instances are instantiated, while *graphical* containment is based on JavaFX elements being added to other JavaFX elements via `getChildren.add(…)` method, and on the resulting scene graph

Outline

- EventHandlers
- Lamda Expressions
- Animation
- Layout Panes
- Java FX Shapes

Animation – Change Over Time

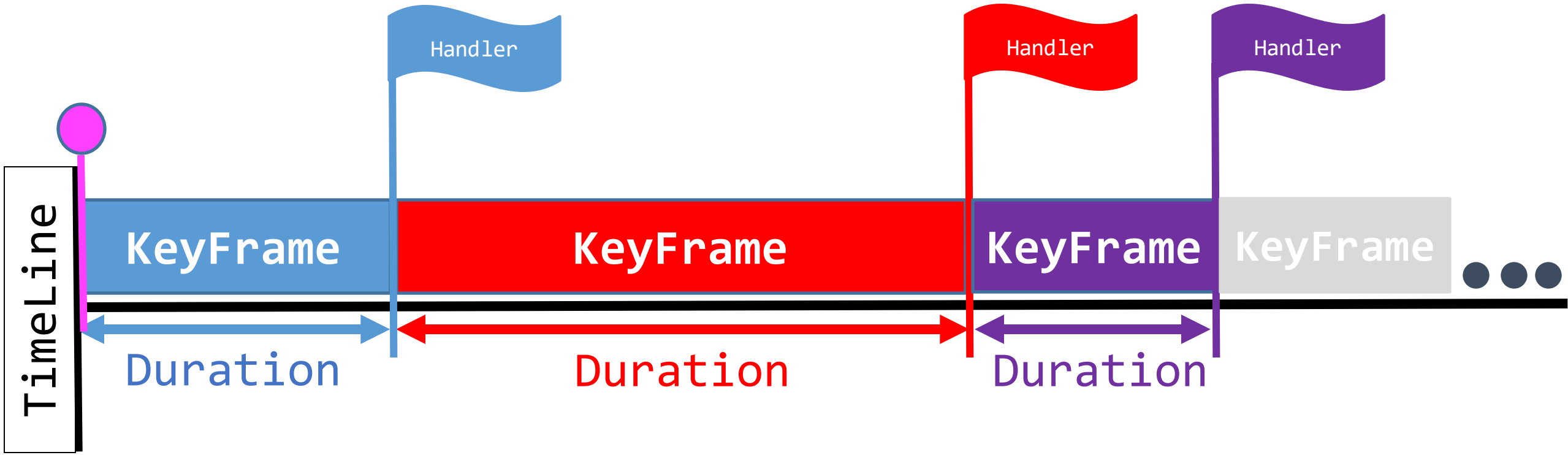
- Suppose we have an alien `Shape` we would like to **animate** (e.g. make it move across the screen)
- As in film and video animation, we can create **apparent motion** with many small changes in position (e.g., Flipbook Animation: https://www.youtube.com/watch?v=ntD2qiGx-DY)
- If we move **fast enough** 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?
  - Use a `Timeline` where we define changes at specific instants



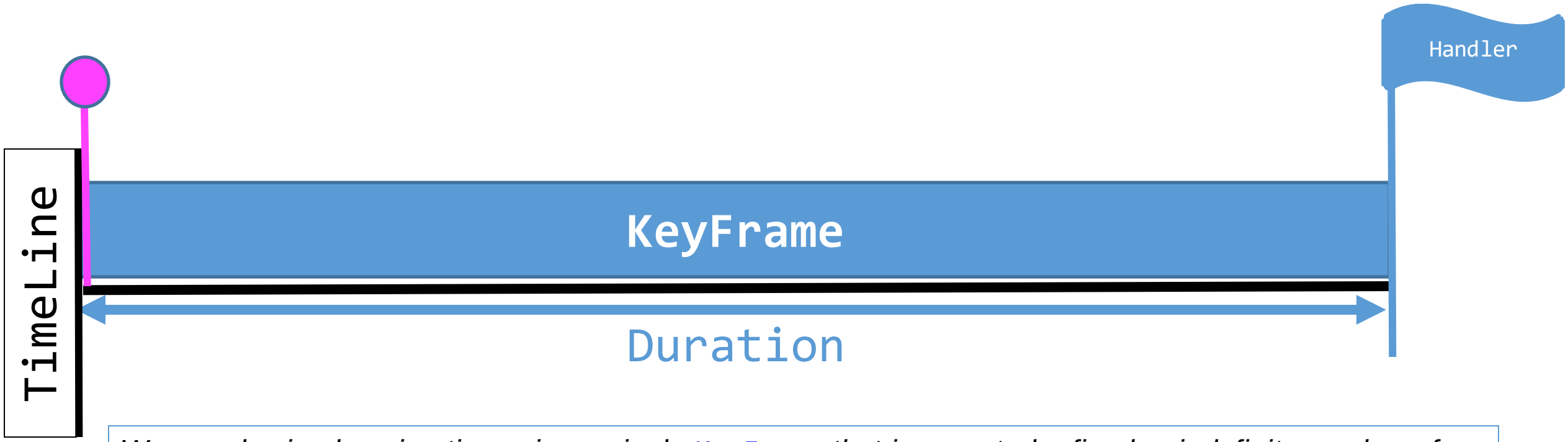
Introducing `Timelines` (1/3)

- The `Timeline` sequences (puts in order) one or more `KeyFrames`
  - a `KeyFrame` can be thought of as a singular snapshot
    - constructed with an associated `Duration` and `EventHandler`
  - in our simple use of JavaFX `KeyFrames`, each 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, `EventHandler` 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 or run through them cyclically
  - each `KeyFrame` lasts for its entire `Duration` until its time interval ends and `EventHandler` is called to make updates
- First, we instantiate a `KeyFrame`, and pass in
  - a `Duration` (e.g. `Duration.seconds(0.3)` or `Duration.millis(300)`), which defines time that each `KeyFrame` lasts
  - an `EventHandler` of type `ActionEvent` 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
- From last lecture: we can use lambda expressions to represent the `EventHandler`s instead of creating a separate class

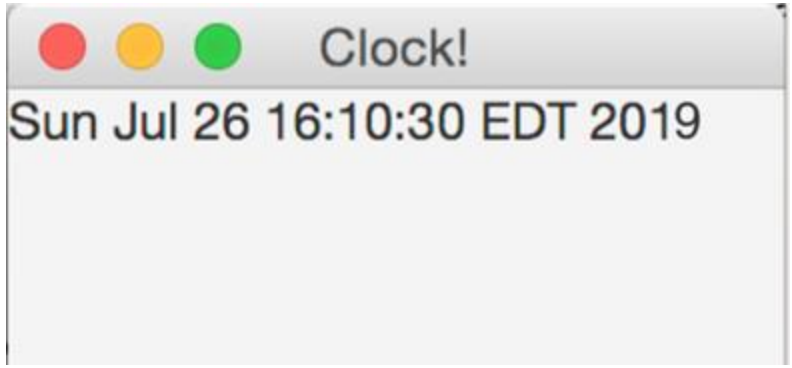
Using JavaFX `Timelines` (2/2)

- Next, we instantiate our `Timeline`, setting its `CycleCount` property
  - defines number of cycles in `Animation`
  - setting `CycleCount` to `Animation.INDEFINITE` will let `Timeline` run forever or until we explicitly stop it
- We pass our new `KeyFrame` into `Timeline`
- After setting up `Timeline`, in order for it to start, we must call

```
timeline.play();
```

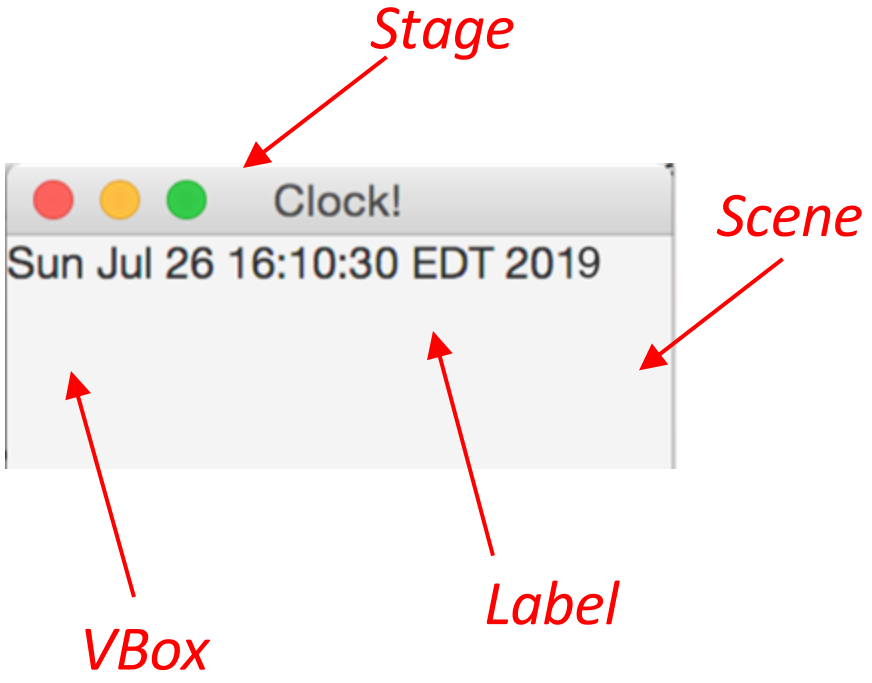
Our First JavaFX animation: `Clock`

- 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 `Duration` and a lambda expression (defined later) as our `EventHandler`. Then instantiate `Timeline`, passing in our `KeyFrame`, and play `Timeline`
4. Define lambda expression to represent our `EventHandler` – for every `ActionEvent`, update the text on the `Label`



Clock: App class (1/3)

Note: Exactly the same process as in ColorChanger's App [Lecture 9]

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(), 300, 200);



    }

}
```

Clock: App class (3/3)

Note: Exactly the same process as in ColorChanger's App [Lecture 9]

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

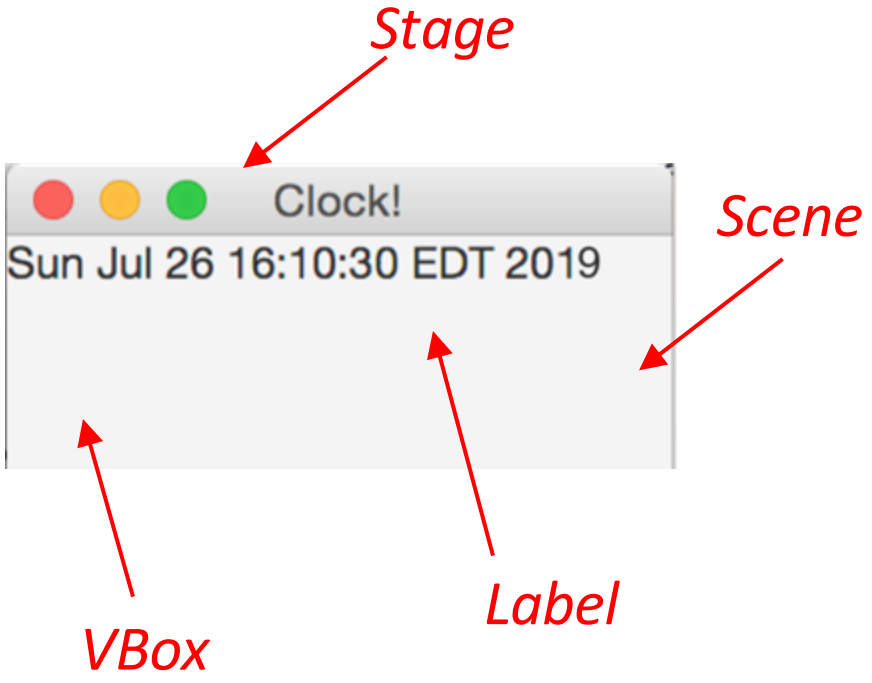
1b. Instantiate a `Scene`, passing in `organizer.getRoot()`, desired width and height of the `Scene`

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(), 300, 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 Duration and a lambda expression (defined later) as our EventHandler. Then instantiate a Timeline, passing in our KeyFrame, and play the Timeline
4. Define lambda expression to represent our EventHandler – for every ActionEvent, update the text on the Label



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() {
            this.root = new VBox();




        }

        public VBox getRoot() {
            return this.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() {
            this.root = new VBox();
            this.label = new Label();
            this.root.getChildren().add(this.label);


        }

        public VBox getRoot() {
            return this.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

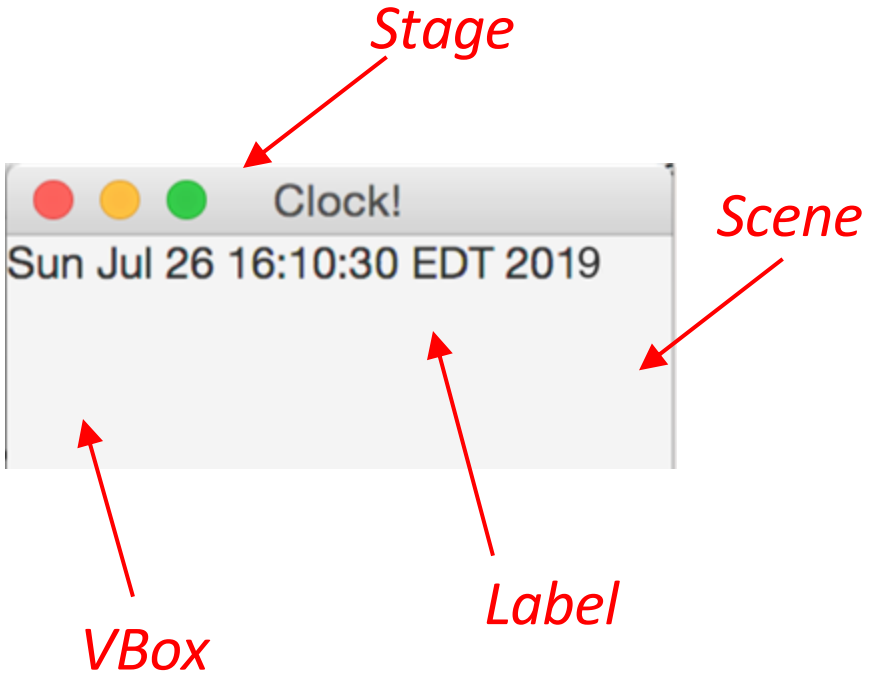
2b. Instantiate a `Label` and add it to the list of the root node's children

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() {
            this.root = new VBox();
            this.label = new Label();
            this.root.getChildren().add(this.label);

            this.setupTimeline();
        }

        public VBox getRoot() {
            return this.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 `Duration` and a lambda expression (defined later) as our `EventHandler`. Then instantiate a `Timeline`, passing in our `KeyFrame`, and play the `Timeline`**
4. Define lambda expression to represent our `EventHandler` – for every `ActionEvent`, update the text on the `Label`



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

Within `setupTimeline()`:

3a. Instantiate a `KeyFrame`, which takes two parameters: Duration and EventHandler

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

    private void setupTimeline() {
        KeyFrame kf = new KeyFrame(
                                          ,

                                        );









    }

}
```

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

Within `setupTimeline()`:

3a. Instantiate a `KeyFrame`, which takes two parameters: Duration and EventHandler

- want to update text of `label` each second – therefore make `Duration` of the `KeyFrame` 1 second

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

    private void setupTimeline() {
        KeyFrame kf = new KeyFrame(
                Duration.seconds(1), //how long

                                    );
    }

}
```

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

Within `setupTimeline()`:

3a. Instantiate a `KeyFrame`, which takes two parameters: Duration and EventHandler

- want to update text of `label` each second – therefore make `Duration` of the `KeyFrame` 1 second
- for the `EventHandler` parameter, pass a lambda expression (to be defined later)

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

    private void setupTimeline() {
        KeyFrame kf = new KeyFrame(
                Duration.seconds(1), //how long
                (ActionEvent e) ->
                this.updateLabel()); //event handler

    }

}
```

Note: JavaFX automatically calls `this.updateLabel` at end of each `KeyFrame`, which in this case changes the label text, and then lets the 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

    private void setupTimeline() {
        KeyFrame kf = new KeyFrame(
                  Duration.seconds(1),
                  (ActionEvent e) ->
                  this.updateLabel()); //event handler

        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

    private void setupTimeline() {
        KeyFrame kf = new KeyFrame(
                  Duration.seconds(1),
                  (ActionEvent e) ->
                  this.updateLabel()); //event handler

        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`

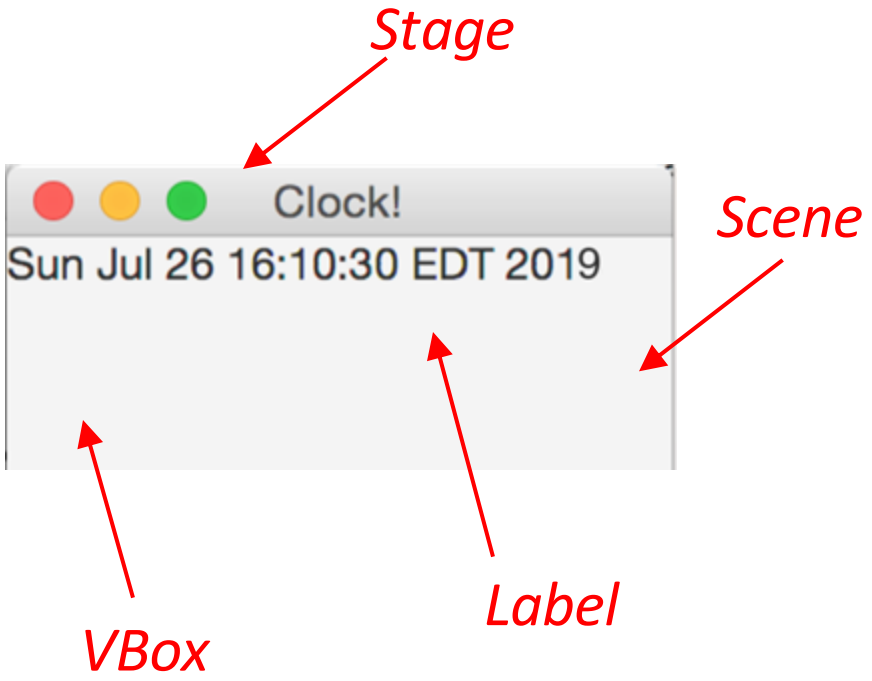
3c. Set `CycleCount` to `INDEFINITE`

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

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

    private void setupTimeline() {
        KeyFrame kf = new KeyFrame(
                  Duration.seconds(1),
                  (ActionEvent e) ->
                  this.updateLabel()); //event handler

        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 a lambda expression (defined later) as our `EventHandler`. Then instantiate a `Timeline`, passing in our `KeyFrame`, and play the `Timeline`
4. **Define a lambda expression to represent our EventHandler – for every ActionEvent, update the text on the Label**



Clock: EventHandler: lambda expression (1/3)

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 JavaFX

```
public class PaneOrganizer {
    private Label label;
        //other code elided

    private void setUpTimeline () {
        KeyFrame kf = new KeyFrame(
                    Duration.seconds(1),
                    (ActionEvent e) ->
                    this.updateLabel()); //event handler
         //other code elided
    }

    private void updateLabel() {









    }
}
```

Clock: EventHandler: lambda expression (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 JavaFX

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 {
    private Label label;
        //other code elided

    private void setUpTimeline () {
        KeyFrame kf = new KeyFrame(
                    Duration.seconds(1),
                    (ActionEvent e) ->
                    this.updateLabel()); //event handler
         //other code elided
    }

    private void updateLabel() {
        Date now = new Date();









    }
}
```

Clock: EventHandler: lambda expression (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 JavaFX

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 {
    private Label label;
        //other code elided

    private void setUpTimeline () {
        KeyFrame kf = new KeyFrame(
                    Duration.seconds(1),
                    (ActionEvent e) ->
                    this.updateLabel()); //event handler
         //other code elided
    }

    private void updateLabel() {
        Date now = new Date();
             //this.label instantiated in
             //constructor of PO
        this.label.setText(now.toString());
    }
}
```

toString( ) converts the Date into a String with year, day, time etc.

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
import javafx.event.ActionEvent;
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(), 300, 200);
            stage.setScene(scene);
            stage.setTitle("Clock");
            stage.show();
        }

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

```
public class PaneOrganizer {
        private VBox root;
        private Label label;

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

        public VBox getRoot() {
            return this.root;
        }

        private void setupTimeline() {
            KeyFrame kf = new KeyFrame(Duration.seconds(1),
                  (ActionEvent e) -> this.updateLabel());
            Timeline timeline = new Timeline(kf);
            timeline.setCycleCount(Animation.INDEFINITE);
            timeline.play();
        }

        private void updateLabel() {
             Date now = new Date();
             this.label.setText(now.toString());
        }

}
```

Outline

- EventHandlers
- Lamda Expressions
- Animation
- Layout Panes
- Java FX Shapes

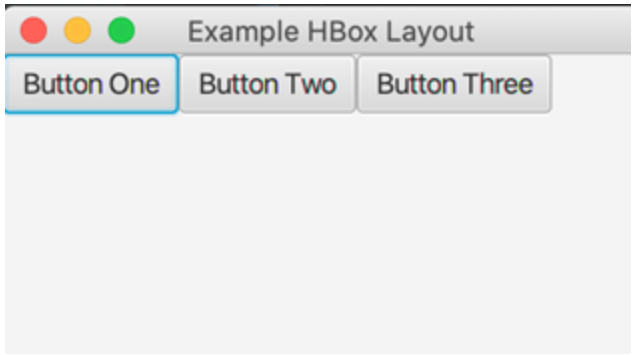
Layout Panes

- Until now, we have been adding all our GUI components to a `VBox`
  - `VBox`es 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);
```

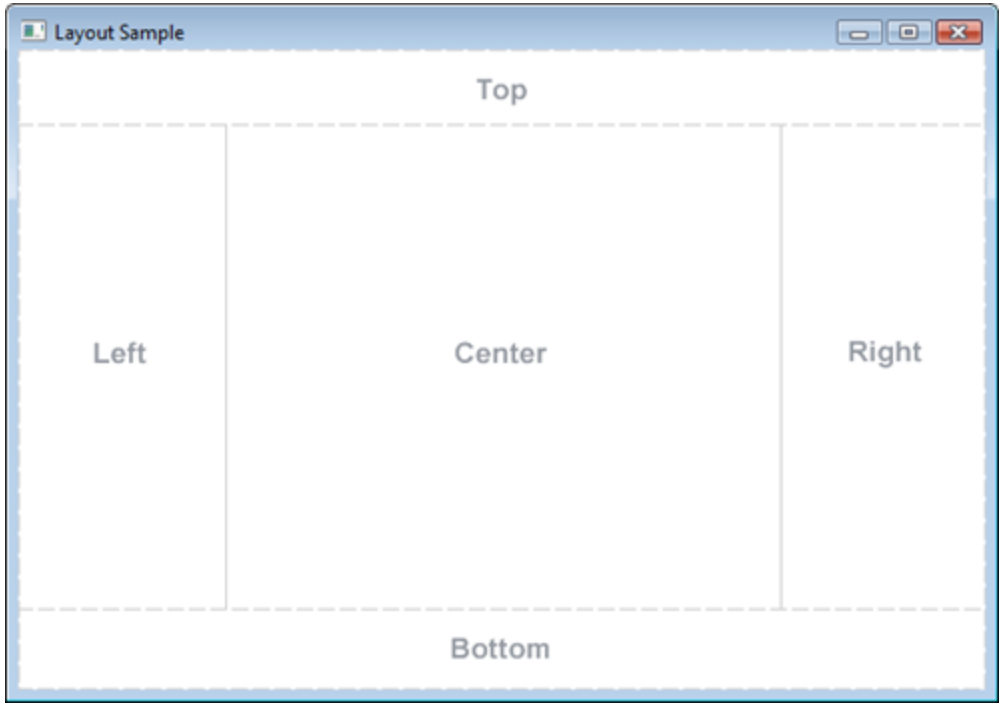


- Like VBox, we can set the amount of horizontal spacing between each child in the HBox using the `setSpacing(double)` method

```
buttonBox.setSpacing(20);
```

`BorderPane` (1/2)

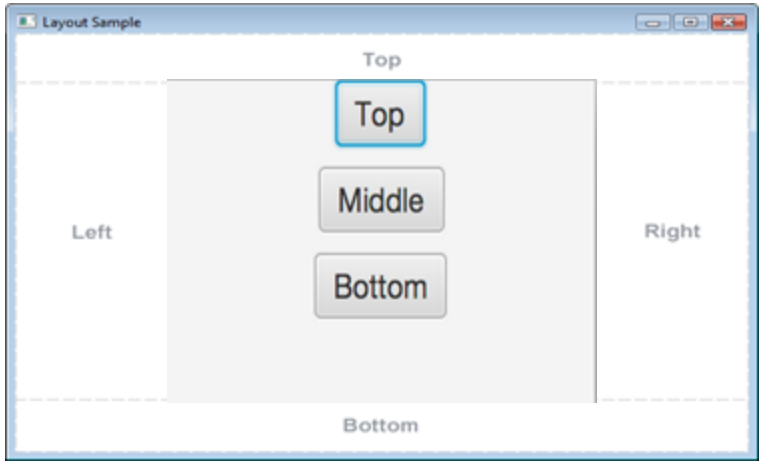
- `BorderPane` lays out children in top, left, bottom, right, and center positions
- To add things visually, use `setLeft(Node)`, `setCenter(Node)`, etc.
  - this includes an implicit call to `getChildren().add(…)`
- Use any type of `Node` – `Pane`s (with their own children), `Button`s, `Label`s, etc.!



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!

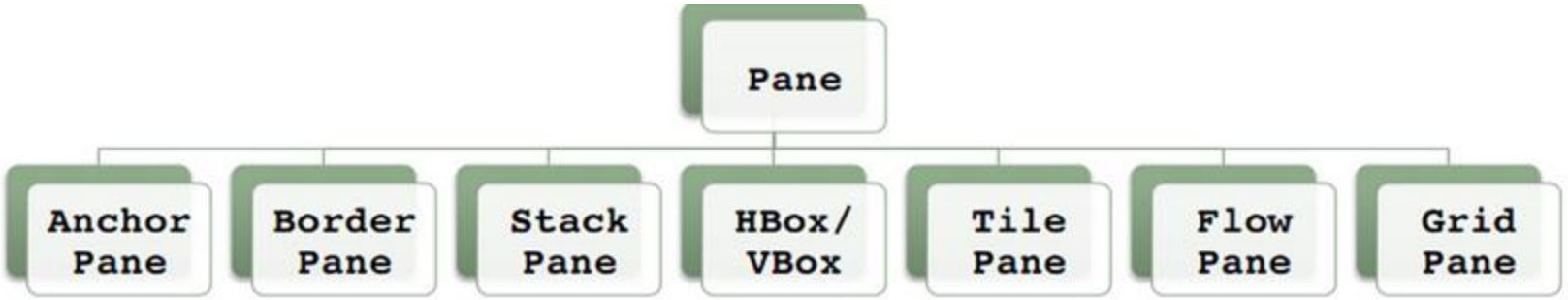
- No need to use all regions – could just use a few of them
- Unused regions are "compressed", e.g. could have a two-region (left/right) layout without a center

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 – DIY graphics! More control, more work 😉
- It is a concrete superclass to all more specialized layout panes seen earlier that do automatic positioning
  - we can call methods on its graphically contained children (panes, buttons, shapes, etc.) to set location within pane
    - for example: use `setX(double)` and `setY(double)` to position a `Rectangle`, one of the primitive shapes
  - `Pane` performs no layout management, so coordinates you `set` determine where things appear on the screen



Creating Custom Graphics

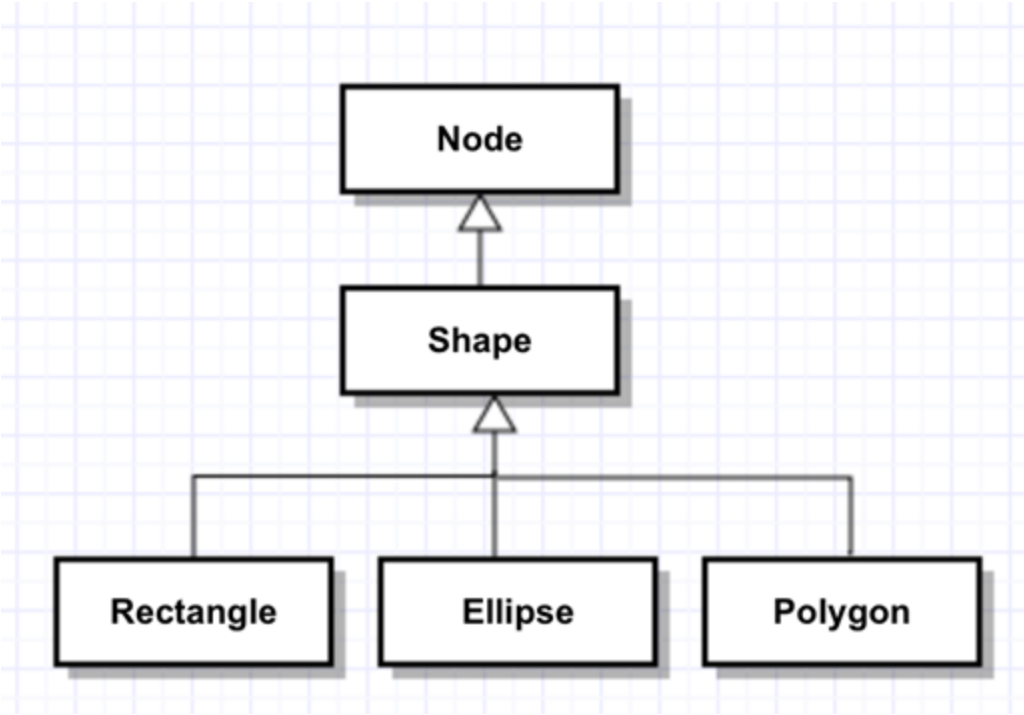
- We've now introduced you to using JavaFX's native UI elements
  - ex: `Label` and `Button`
- Lots of handy widgets for making your own graphical applications!
- What if you want to create your own custom graphics?
- This lecture: build your own graphics using the `javafx.scene.shape` package!

Outline

- EventHandlers
- Lamda Expressions
- Animation
- Layout Panes
- Java FX Shapes

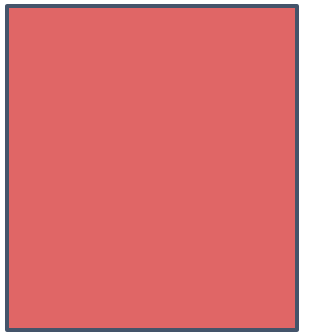
`javafx.scene.shape` Package

- 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.

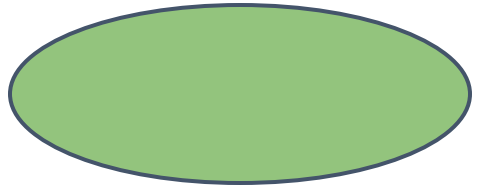


Shape Constructors

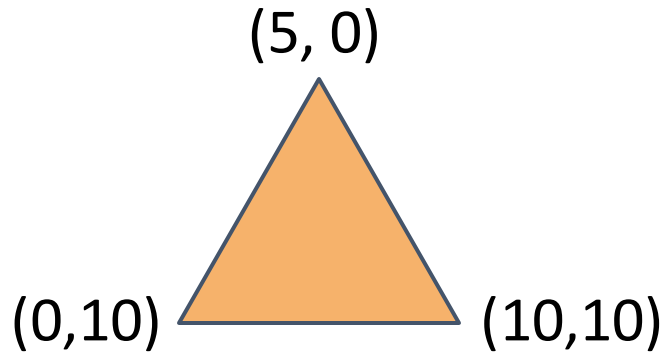
Default position for Shape with this constructor would be (0,0)



- `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 `Shapes` (like `Rectangle` or `Ellipse`)
  - example: `new Polygon(0,10,10,10,5,0)`

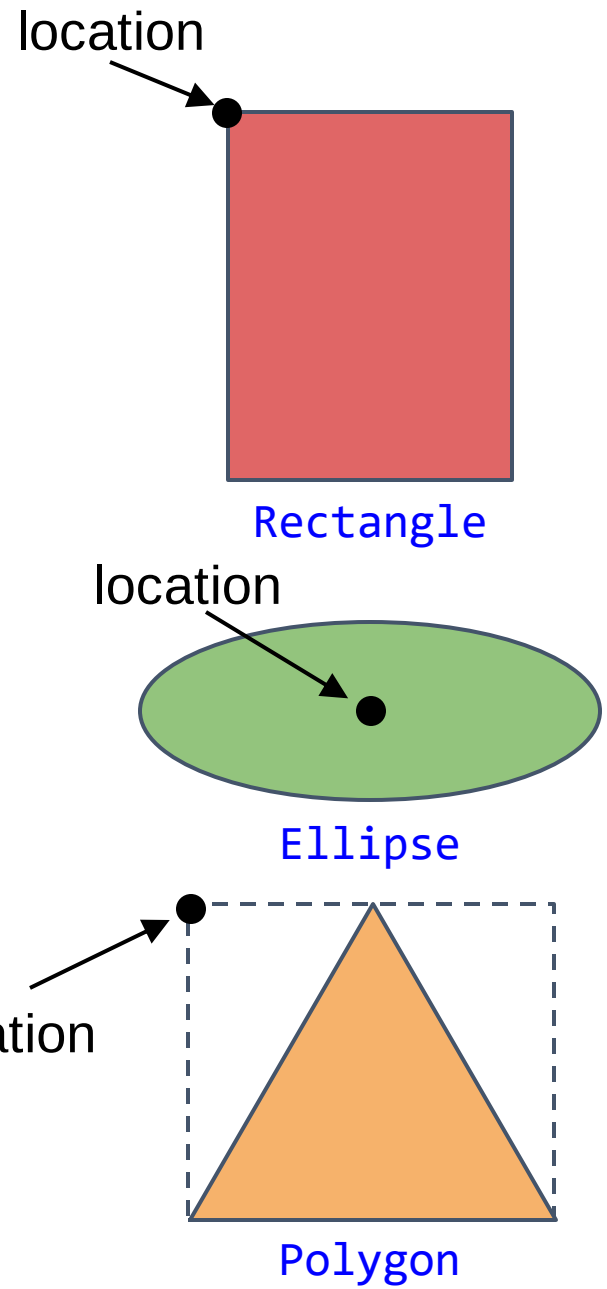


- Each of these `Shape` subclasses have multiple constructors (same name, different parameter lists) This is called method overloading – we'll come back to it during Design Patterns. Check out the JavaFX documentation for more options!
  - for example, if you wanted to instantiate a `Rectangle` with a given position and size:
    `Rectangle(double x, double y, double width, double height)`
  - you could also instantiate a Rectangle with a given width, height, and color:
    `Rectangle(double width, double height, Paint fill)`



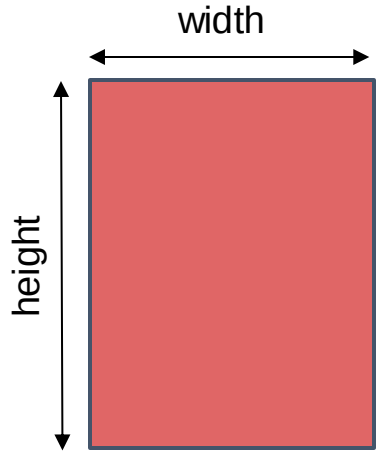
Shapes: Setting Location

- 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!

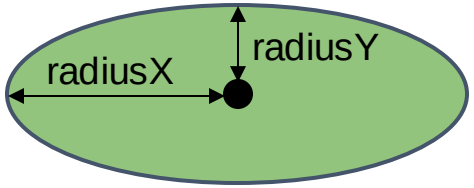


Shapes: Setting Size

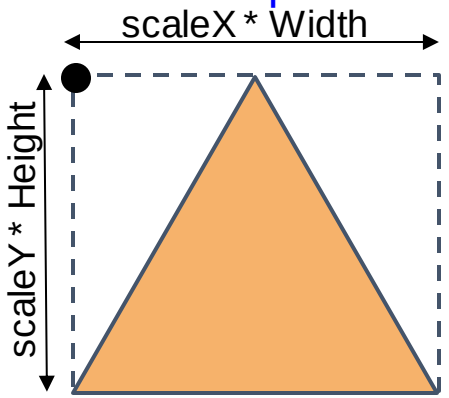
- JavaFX Shapes also have different behaviors (methods) for altering their size
  - Rectangle: use setWidth(double) and setHeight(double)
  - Ellipse: use setRadiusX(double) and setRadiusY(double)
  - Polygon: use setScaleX(double) and setScaleY(double)
    - multiplies the original size in the X or 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!



Rectangle



Ellipse



Polygon

Accessors and Mutators of all Shapes

- Setters and Getters!

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 with a new center of rotation to the Shape's transform list (see Javadocs)

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

Generally, use 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 (1/2)

- Code from today's lecture is available on GitHub – mess around for practice!
- Fruit Ninja deadlines (all due 11:59 PM ET):
  - On-time handin: **today 10/11**
  - Late handin: **Thursday 10/13**
- Java FX Lab
  - Pre-lab video and pre-lab quiz
- Fill out the GitHub Username Form
- Fruit Ninja Code Debriefs coming up!
  - Keep an eye on your emails to see if you were chosen as tribute!
  - Not an exam! Just a chance to talk though **YOUR** implementation ☺

Announcements (2/2)

- Collaboration Policy Phase 2 starting at Cartoon
  - can debug each other's terminal-produced errors
  - fill out mandatory collaboration phase 2 quiz

	Phase 1 Debugging Policy	Phase 2 Debugging Policy
Discuss lecture material and general concepts	✓	✓
Collaborate on mini-assignments and labs	✓	✓
Get help from TAs at TA Hours and on Ed	✓	✓
Discuss high-level project-specific concepts and all material provided in handouts and section	✓	✓
Help another student debug a terminal-produced error message, as long as your own computer is closed	✗	✓
Help another student debug a logical code error	✗	✗
Share or compare code with another student	✗	✗
Discuss project-specific implementation details	✗	✗


PONG
TETRIS
JUMP

Option 1

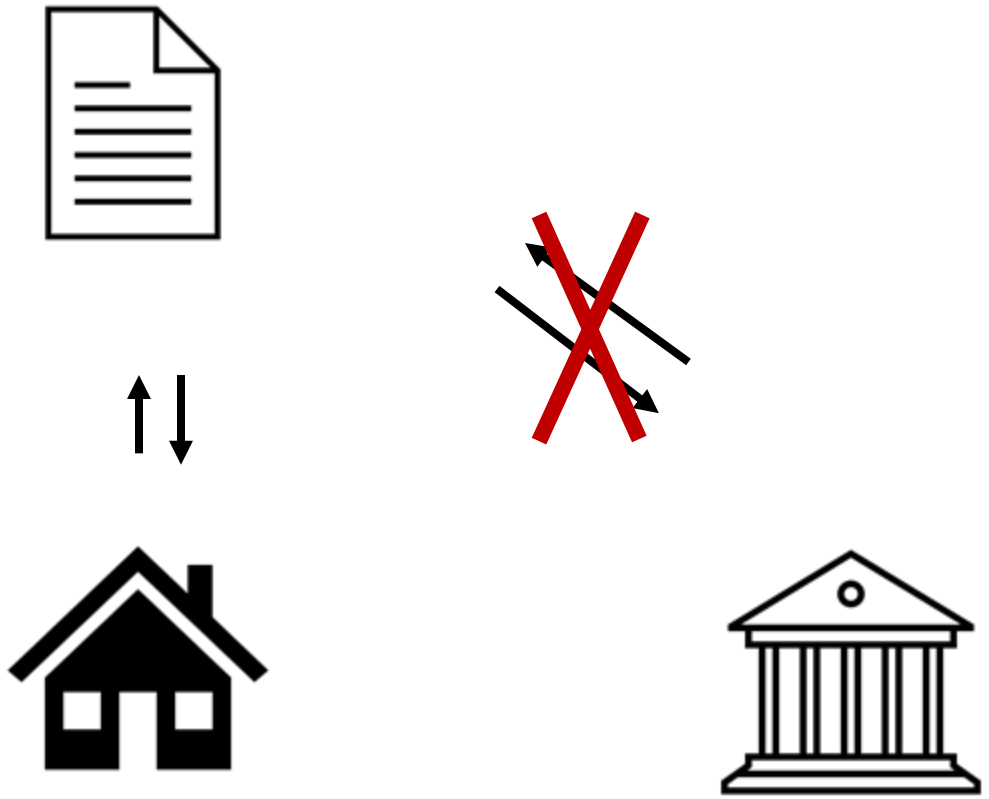
Option 2

Socially Responsible Computing

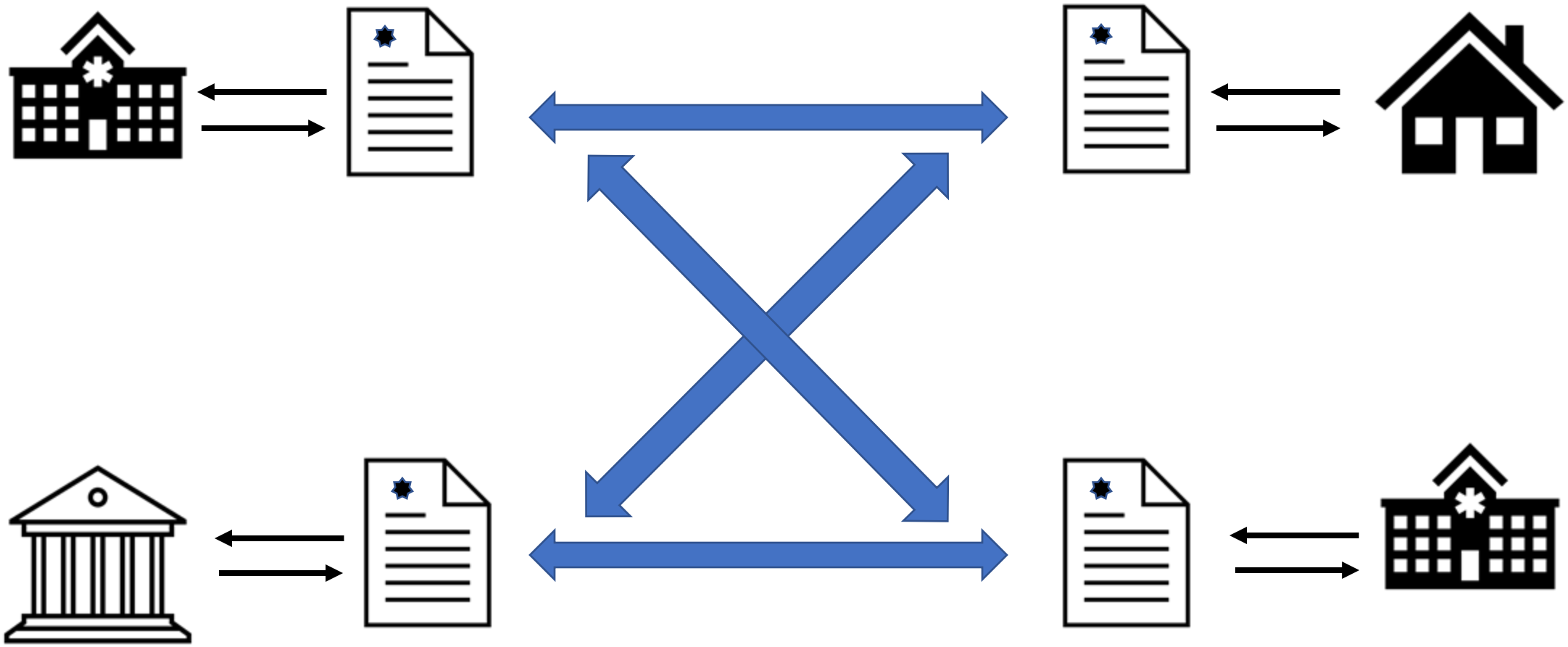
Blockchain & Cryptocurrency I

CS15 Fall 2023

The Status Quo: Centralized Databases



The Utopian Promise: An interoperable, decentralized database that maintains the privacy of users



Introduction to Blockchain Tech

Picture a massive excel spreadsheet that records transactions but make it…

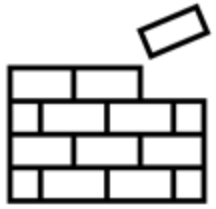
	A	B	C
1	**Last Name**	**Sales**	**Product Type**
2	Smith	$1,675.00	EEE-312
3	Johnson	$1,480.00	DC-1
4	Williams	$1,064.00	EE-2
5	Jones	$1,390.00	DF-3
6	Brown	$4,865.00	EEE-45
7	Williams	$1,243.00	FD-2
8	Johnson	$9,339.00	DC-1

Image source: Excel Easy

Duplicated across a vast network of computers

Raw data is public and open-access

Each transaction and identities are encrypted



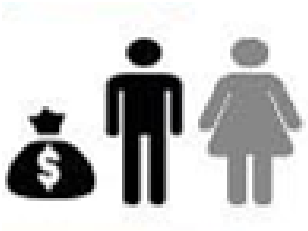
Append-only, changes are permanent

Regularly updated

… which results in a ginormous, decentralized ledger that allows us to verify the validity of future transactions

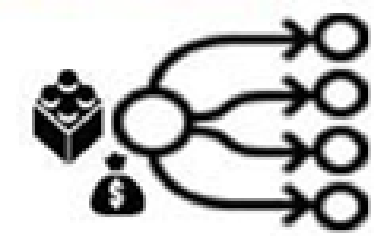
How Money Transfers Over Blockchain Work

Jim wants to send money to Mary

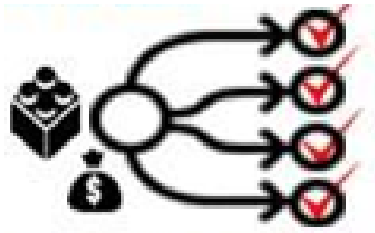


The transaction is represented as a block

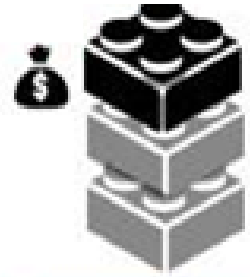
The block gets distributed across the network



The network verifies the transaction is valid



The block is added to the chain, creating a permanent record



Jim’s record of ownership of the money moves to Mary

Original Image Source: Paul Dughi

Economic philosophy of Silicon Valley

Source: Stanford Business (2017)

Why decentralization?

- Attractive to libertarian viewpoint
- Free from government oversight; governed by users


Inc.

NEWSLETTERS SUBSCRIBE

TECHNOLOGY

Peter Thiel Says, 'Crypto Is Libertarian, A.I. Is Communist.' What the Heck Does That Mean? Bonus: Why you

Cryptocurrency: a digital currency in which transactions are verified and records are maintained by a decentralized system

- Born out of the 2008 financial crisis

Source: Inc (2018)

Collapse of FTX

Feb 2022 Super Bowl Commercial

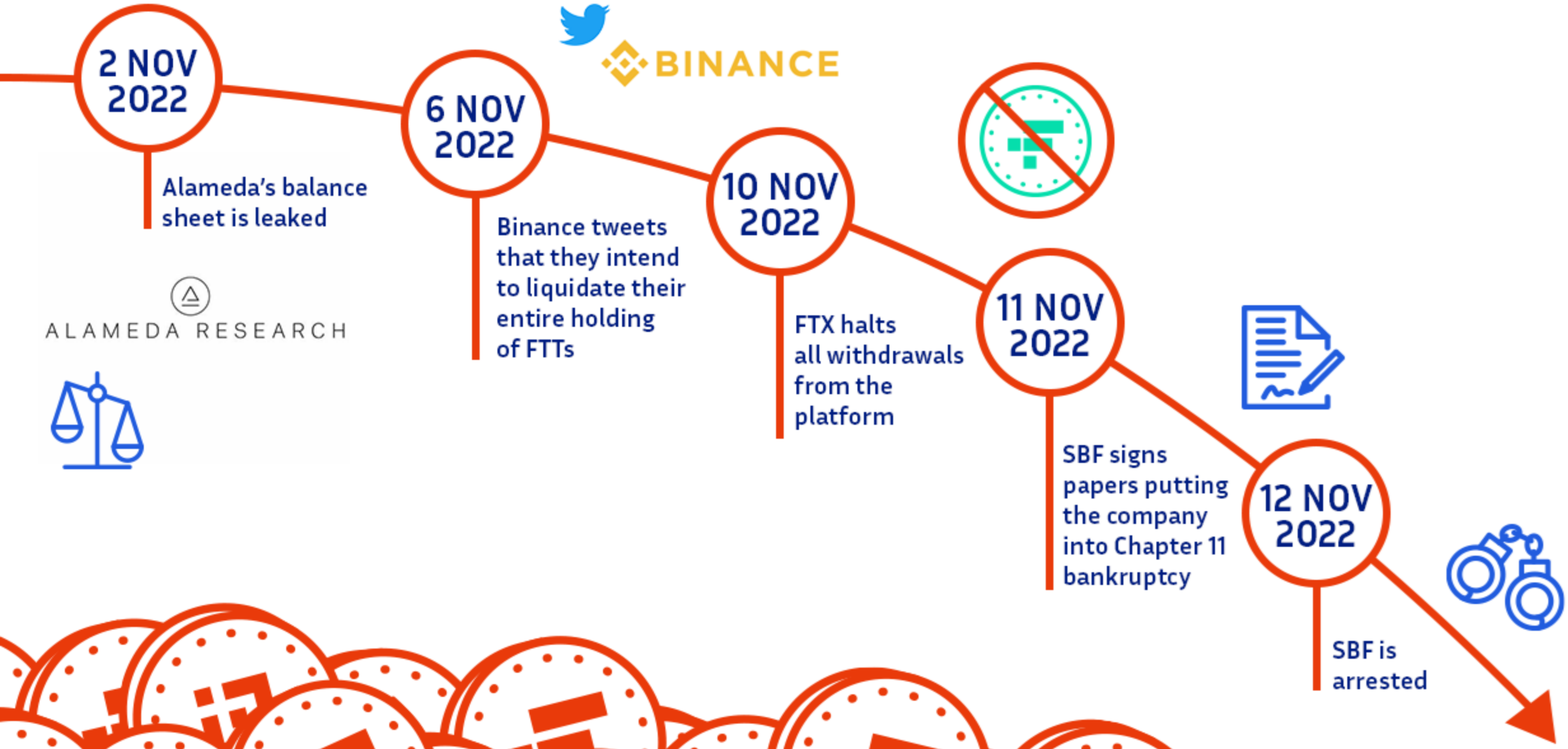
CURRENCIES | CRYPTOCURRENCY

FTX Tapped Into Customer Accounts to Fund Risky Bets, Setting Up Its Downfall

FTX's chief executive told investors this week that an affiliated trading firm owes the crypto exchange about $10 billion

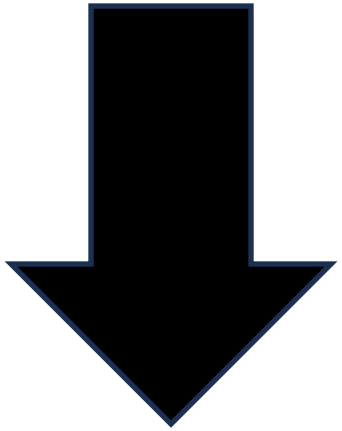
Nov 2022 Wall Street Journal

Collapse of FTX



Source: StructureFlow

Collapse of FTX



$152 Billion

decrease in world's 15 largest cryptocurrencies between 11/8/22 – 11/11/22

Source: WSJ