Graphics Part II
Outline

• Layout Panes
• Absolute Positioning
• Shapes
  o example: MovingShape
• Constants
• Composite Shapes
  o example: MovingAlien
Layout Panes

• Until now, we have been adding all our GUI components to a VBox
  o VBoxes lay everything out in one vertical column
• What if we want to make some more interesting GUIs?
• Use different types of layout panes!
  o VBox is just one of many JavaFX panes—there are many more options
  o 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:

```java
// 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.

![Layout Sample](image)
BorderPane (2/2)

- Remember our VBox example from earlier?

```java
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

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

- No need to use all of the regions—you could just use a few of them

- Unused regions are “compressed”, e.g. you could have a two-region (left/right) layout without a center

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

- Until now, all the layout panes we have seen have performed layout management for us
  - What if we want to position our GUI components freely ourselves?
- We need to set the component’s location to the exact *pixel location* on the screen
  - Called *absolute positioning*
- When would you use this?
  - To position shapes—stay tuned!
Pixels and Coordinate System

- Screen is a grid of **pixels** (tiny squares, each with RGB components)
- Cartesian plane with:
  - origin in upper-left corner
  - x-axis increasing left to right
  - y-axis increasing top to bottom
  - corresponds to English writing order
Pane

- Pane allows you to layout things completely freely—like on an art canvas
- It is a concrete superclass to all of the 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
  o 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
  o Methods relating to rotation and visibility are defined in Node
  o Methods relating to color and border are defined in Shape
  o Other methods are implemented in the individual classes of Ellipse, Rectangle, etc.
Shape Constructors

- Rectangle(double width, double height)
- Ellipse(double radiusX, double radiusY)
- Polygon(double … points)
  - 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

Each of these Shape subclasses have multiple constructors, check out the JavaFX documentation for more options!
  - For example, if you wanted instantiate a Rectangle with a given position and size: Rectangle(double x, double y, double width, double height)

Example: `new Polygon(0,10,10,10,5,0)`
**Shapes: Setting Location**

- JavaFX **Shapes** have different behavior 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)`
    - this is an inherited method, **Polygon** does not implement its own specific method for setting location unlike **Rectangle** and **Ellipse**!

- JavaFX has *many* different ways to set location
  - from our experience, this is the most straightforward way
  - if you choose to use other methods, be sure you fully understand them or you may get strange bugs!
  - check out our [JavaFX documentation](https://docs.oracle.com/javase/8/javafx/api/) and the [Javadoc](https://docs.oracle.com/javase/8/javafx/api/) for more detailed explanations!
**Shapes: Setting Size**

- JavaFX *Shapes* also have different behavior for setting 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 an inherited method – *Polygon* does not implement its own specific method for setting size

- Again, this is not the only way to set size for *Shapes* but it is relatively painless
  - reminder: [JavaFX documentation](https://docs.oracle.com/javase/8/javafx/api/) and [Javadoc](https://docs.oracle.com/javase/8/javafx/api/)!
Accessors and Mutators of all **Shapes**

- **Rotation:**
  - public final void setRotate(double rotate);
  - 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 getStokeWidth();

** 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);**

- **Rotation is about the center of the Shape**
- **The stroke is the border that outlines the Shape, while the fill is the color of the interior of the Shape**

**final = can't override method**
Example: **MovingShape**

- Spec: App that displays a shape and buttons that shift position of the shape left and right by a fixed increment
- Practice working with absolute positioning of Panes, various Shapes, & more event handling!
Process: **MovingShapeApp**

1. **Write a top-level App class that extends javafx.application.Application and implements start (standard pattern)**

2. Write a `PaneOrganizer` class that instantiates root node and makes a public `getRoot()` method. In `PaneOrganizer`, create an `Ellipse` and add it as child of root `Pane`.

3. Write `setupShape()` and `setupButtons()` helper methods to be called within `PaneOrganizer`’s constructor. These will factor out the code for creating our custom `Pane`.

4. Register `Buttons` with `EventHandlers` that handle `Buttons’ ActionEvents (clicks)` by moving `Shape` correspondingly.
(1) Top-level Class: MovingShapeApp

*NOTE: Exactly the same process as previous examples*

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

```java
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("Color Changer");
        stage.show();
    }
}
```
(1) **Top-level Class: MovingShapeApp**

*NOTE: Exactly the same process as previous examples*

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

B. **Instantiate a Scene, passing in** `organizer.getRoot()` **and desired width and height of Scene**

```java
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("Color Changer");
        stage.show();
    }
}
```
(1) Top-level Class: `MovingShapeApp`

*NOTE: Exactly the same process as previous examples*

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

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

C. **Set scene, set Stage’s title and show it!**

```java
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("Moving Shape!");
        stage.show();
    }
}
```
Process: **MovingShapeApp**

1. Write a top-level `App` class that extends `javafx.application.Application` and implements `start` (standard pattern)

2. Write a `PaneOrganizer` class that instantiates root node and makes a public `getRoot()` method. In `PaneOrganizer`, create an `Ellipse` and add it as child of root `Pane`

3. Write `setupShape()` and `setupButtons()` helper methods to be called within `PaneOrganizer`'s constructor. These will factor out the code for creating our custom `Pane`

4. Register `Buttons` with `EventHandlers` that handle `Buttons' ActionEvents` (clicks) by moving `Shape` correspondingly
(2) PaneOrganizer Class

A. Instantiate the root Pane and store it in the instance variable _root

```java
public class PaneOrganizer {
    private Pane _root;

    public PaneOrganizer() {
        _root = new Pane();
    }
}
```
(2) PaneOrganizer Class

A. Instantiate the root Pane and store it in the instance variable _root

B. Create a public getRoot() method that returns _root

```java
public class PaneOrganizer {
    private Pane _root;

    public PaneOrganizer() {
        _root = new Pane();
    }

    public Pane getRoot() {
        return _root;
    }
}
```
(2) **PaneOrganizer** Class

A. Instantiate the root Pane and store it in the instance variable `_root`

B. Create a public `getRoot()` method that returns `_root`

C. Instantiate the Ellipse and add it as child of the root Pane

```java
public class PaneOrganizer {
    private Pane _root;
    private Ellipse _ellipse;

    public PaneOrganizer() {
        _root = new Pane();
        _ellipse = new Ellipse(50, 50);
        _root.getChildren().add(_ellipse);
    }

    public Pane getRoot() {
        return _root;
    }
}
```
(2) **PaneOrganizer** Class

A. Instantiate the root Pane and store it in the instance variable _root

B. Create a public `getRoot()` method that returns _root

C. Instantiate the Ellipse and the Buttons, and add them as children of the root Pane

D. Call `setupShape()` and `setupButtons()`, defined next

```java
public class PaneOrganizer {
    private Pane _root;
    private Ellipse _ellipse;

    public PaneOrganizer() {
        _root = new Pane();
        _ellipse = new Ellipse(50, 50);
        _root.getChildren().add(_ellipse);

        this.setupShape();
        this.setupButtons();
    }

    public Pane getRoot() {
        return _root;
    }
}
```
Process: **MovingShapeApp**

1. Write a top-level `App` class that extends `javafx.application.Application` and implements `start` (standard pattern)

2. Write a `PaneOrganizer` class that instantiates root node and makes a public `getRoot()` method. In `PaneOrganizer`, create an `Ellipse` and add it as child of root `Pane`

3. Write `setupShape()` and `setupButtons()` helper methods to be called within `PaneOrganizer`’s constructor. These will factor out the code for creating our custom `Pane`

4. Register `Buttons` with `EventHandlers` that handle `Buttons' ActionEvents` (clicks) by moving `Shape` correspondingly
(3) PaneOrganizer Class

• As our applications start getting more complex, we will need to write a lot more code to get the UI looking the way we would like
• Such code would convolute the PaneOrganizer constructor—it is good practice to factor out code into helper methods that are called within the constructor
• setupShape() fills and positions Ellipse
• setupButtons() adds and positions Buttons, and registers them with their appropriate EventHandlers
(3) **PaneOrganizer**: `setupShape()` helper method

- For this application, “helper method” `setupShape()` will only set fill color and position Ellipse in Pane using absolute positioning
- Helper method is `private`—why is this good practice?
  - only the `PaneOrganizer` should be allowed to initialize the color and location of the Ellipse
  - `private` methods are not directly inherited and are not accessible to subclasses—though inherited methods may make use of them!
  - generally, helper methods should be `private`
Digression: PaneOrganizer Class

• We were able to absolutely position _ellipse in the root Pane because our root is simply a Pane and not one of the more specialized subclasses

• We could also use absolute positioning to position the Buttons in the Pane in our setUpButtons() method...But look how annoying trial-and-error is!

Is there a better way? ...hint: leverage Scene Graph hierarchy and delegation!
**Digression: PaneOrganizer Class**

- Rather than absolutely positioning **Buttons** directly in root **Pane**, use a specialized layout **Pane**: add a new **HBox** as a child of the root **Pane**
  - add **Buttons** to **HBox**, to align horizontally
- Continuing to improve our design, use a **BorderPane** as root
- Now need to add need **Ellipse** to the root
  - could simply add **Ellipse** to center of root
  - but this won’t work—if we let the **BorderPane** dictate the placement of the **Ellipse** we won’t be able to update the position of it using the **Buttons**
  - instead: create a **Pane** to contain **Ellipse** and add the **Pane** as child of root!
Digression: PaneOrganizer Class

• This makes use of the built-in layout capabilities available to us in JavaFX!

• Note: this is only one of *many* design choices for this application!
  o Keep in mind all of your different layout options when designing your programs!
  o Absolutely positioning the entire program is most likely *not* the best solution
(3) PaneOrganizer Class: update to BorderPane

A. Change root to a BorderPane, create a Pane to contain Ellipse

```java
public class PaneOrganizer {
    private BorderPane _root;
    private Ellipse _ellipse;

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

        // setup shape pane
        Pane shapePane = new Pane();
        _ellipse = new Ellipse(50, 50);
        shapePane.getChildren().add(_ellipse);

        this.setupShape();
        this.setupButtons();
    }

    private void setupButtons() {
        // more code to come!
    }
}
```
(3) **PaneOrganizer** Class: update to **BorderPane**

A. Change root to a `BorderPane`, create a `Pane` to contain `Ellipse`  

B. To add `shapePane` to center of `BorderPane`, call `setCenter(shapePane)` on `root`  
   - note: none of the code in our `setupShape()` method needs to be updated – our `Ellipse` is just graphically contained within a different `Pane`  
   - but `PaneOrganizer` still can access to it!

```java
public class PaneOrganizer {
    private BorderPane _root;
    private Ellipse _ellipse;

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

        // setup shape pane
        Pane shapePane = new Pane();
        _ellipse = new Ellipse(50, 50);
        shapePane.getChildren().add(_ellipse);
        _root.setCenter(shapePane);
        this.setupShape();
        this.setupButtons();
    }

    private void setupButtons() {
        // more code to come!
    }
}
```
(3) PaneOrganizer Class: `setupButtons()` method

C. Instantiate a new HBox, then add it as a child of BorderPane, in bottom position

```java
public class PaneOrganizer {
    private BorderPane _root;
    private Ellipse _ellipse;

    public PaneOrganizer() {
        _root = new BorderPane();
        // setup of shape pane and shape elided!
        this.setupButtons();
    }

    private void setupButtons() {
        Hbox buttonPane = new Hbox();
        _root.setBottom(buttonPane);
    }
}
```
(3) **PaneOrganizer** Class: **setupButtons()** method

C. Instantiate a new **HBox**, then add it as a child of **BorderPane**, in bottom position

D. **Instantiate two Buttons**

```java
public class PaneOrganizer {
    private BorderPane _root;
    private Ellipse _ellipse;

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

        // setup of shape pane and shape elided!
        this.setupButtons();
    }

    private void setupButtons() {
        Hbox buttonPane = new Hbox();
        _root.setBottom(buttonPane);
        Button b1 = new Button("Move Left");
        Button b2 = new Button("Move Right");
    }
}
```
(3) PaneOrganizer Class: setupButtons() method

C. Instantiate a new HBox, then add it as a child of BorderPane, in bottom position

D. Instantiate two Buttons

E. Add the Buttons as children of the new HBox
   o note: different from before — now adding Buttons as children of HBox
   o remember that a Node, like a Button, can have at most one parent!

```java
public class PaneOrganizer {
    private BorderPane _root;
    private Ellipse _ellipse;

    public PaneOrganizer() {
        _root = new BorderPane();
        // setup of shape pane and shape elided!
        this.setupButtons();
    }

    private void setupButtons() {
        Hbox buttonPane = new Hbox();
        _root.setBottom(buttonPane);
        Button b1 = new Button("Move Left");
        Button b2 = new Button("Move Right");
        buttonPane.getChildren().addAll(b1, b2);
    }
}
```
(3) PaneOrganizer Class: setupButtons() method

F. Set horizontal spacing between Buttons as you like

```java
public class PaneOrganizer {
    private BorderPane _root;
    private Ellipse _ellipse;

    public PaneOrganizer() {
        _root = new BorderPane();
        // setup of shape pane and shape elided!
        this.setupButtons();
    }

    private void setupButtons() {
        Hbox buttonPane = new Hbox();
        _root.setBottom(buttonPane);
        Button b1 = new Button("Move Left");
        Button b2 = new Button("Move Right");
        buttonPane.getChildren().addAll(b1, b2);
        buttonPane.setSpacing(30);
    }
}
```
(3) PaneOrganizer Class: setupButtons() method

F. Set horizontal spacing between Buttons as you like

G. Register Buttons with their EventHandlers by calling setOnAction() and passing in our instances of MoveHandler, which we will create next!
**Process: MovingShapeApp**

1. Write a top-level App class that extends `javafx.application.Application` and implements `start` (standard pattern)

2. Write a `PaneOrganizer` class that instantiates root node and makes a public `getRoot()` method. In `PaneOrganizer`, create an `Ellipse` and add it as child of root `Pane`

3. Write `setupShape()` and `setupButtons()` helper methods to be called within `PaneOrganizer`’s constructor. These will factor out the code for creating our custom `Pane`

4. Register **Buttons** with **EventHandlers** that handle **Buttons’ ActionEvents** (clicks) by moving **Shape** correspondingly
**Digression: Creating EventHandlers**

- Our goal is to register each button with an EventHandler
  - the “Move Left” Button moves the Ellipse left by a set amount
  - the “Move Right” Button moves the Ellipse right the same amount

- We could define two separate EventHandlers, one for the “Move Left” Button and one for the “Move Right” Button…
  - why might this not be the optimal design?
  - remember, we want to be efficient with our code usage!

- Instead, we can define one EventHandler
  - factor out common behavior into one class
  - specifics determined by parameters passed into the constructor!
(4) **MoveHandler**: Private Inner Class

A. Declare an instance variable \_distance that will be initialized differently depending on whether the isLeft argument is true or false

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

    public PaneOrganizer() {
        // other code elided
    }

    private class MoveHandler implements EventHandler<ActionEvent> {
        private int _distance;

        public MoveHandler(boolean isLeft) {

        }

        public void handle(ActionEvent e) {

        }
    }
}
```
(4) **MoveHandler**: Private Inner Class

A. Declare an instance variable `_distance` that will be initialized differently depending on whether the `isLeft` argument is **true** or **false**

B. **Set _distance to 10 initially** – if the registered Button `isLeft`, **change _distance to -10** so the Ellipse moves in the opposite direction

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

    public PaneOrganizer() {
        // other code elided
    }

    private class MoveHandler implements EventHandler<ActionEvent> {
        private int _distance;

        public MoveHandler(boolean isLeft) {
            _distance = 10;
            if (isLeft) {
                _distance *= -1; // change sign
            }
        }

        public void handle(ActionEvent e) {
            // handle
        }
    }
}
```
(4) **MoveHandler**: Private Inner Class

A. Declare an instance variable `_distance` that will be initialized differently depending on whether the `isLeft` argument is true or false

B. Set `_distance` to 10 initially – if the registered Button is `isLeft`, change `_distance` to -10 so the Ellipse moves in the opposite direction

C. Implement the handle method to move the Ellipse by `_distance` in the horizontal direction

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

    public PaneOrganizer() {
        // other code elided
    }

    private class MoveHandler implements EventHandler<ActionEvent> {
        private int _distance;

        public MoveHandler(boolean isLeft) {
            _distance = 10;
            if (isLeft) {
                _distance *= -1; // change sign
            }
        }

        public void handle(ActionEvent e) {
            _ellipse.setCenterX(_ellipse.getCenterX() + _distance);
        }
    }
}
```
The Whole App

package MovingShape;

// imports for the App class
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

// imports for the PaneOrganizer class
import javafx.event.*;
import javafx.geometry.Pos;
import javafx.scene.control.Button;
import javafx.scene.layout.*;
import javafx.scene.paint.Color;
import javafx.scene.shape.Ellipse;

public class PaneOrganizer {
    private BorderPane _root;
    private Ellipse _ellipse;

    public PaneOrganizer() {
        _root = new BorderPane();
        Pane shapePane = new Pane();
        _ellipse = new Ellipse(50, 50);
        shapePane.getChildren().add(_ellipse);
        _root.setCenter(shapePane);
        this.setupShape();
        this.setupButtons();
    }

    public Pane getRoot() {
        return _root;
    }

    private void setupShape() {
        _ellipse.setFill(Color.RED);
        _ellipse.setCenterX(100);
        _ellipse.setCenterY(50);
    }

    private void setupButtons() {
        HBox buttonPane = new HBox();
        _root.setBottom(buttonPane);
        Button b1 = new Button("Move Left!");
        Button b2 = new Button("Move Right!");
        buttonPane.getChildren().addAll(b1, b2);
        buttonPane.setSpacing(30);
        buttonPane.setAlignment(Pos.CENTER);
        b1.setOnAction(new MoveHandler(true));
        b2.setOnAction(new MoveHandler(false));
    }

    private class MoveHandler implements EventHandler<ActionEvent> {
        private int _distance;

        public MoveHandler(boolean isLeft) {
            _distance = 10;
            if (isLeft) {
                _distance *= -1;
            }
        }

        public void handle(ActionEvent event) {
            _ellipse.setCenterX(_ellipse.getCenterX() + _distance);
        }
    }
}

// end of private MoveHandler class

public class App extends Application {
    @Override
    public void start(Stage stage) {
        PaneOrganizer organizer = new PaneOrganizer();
        Scene scene = new Scene(organizer.getRoot(), 200, 130);
        stage.setScene(scene);
        stage.setTitle("MovingShape!");
        stage.show();
    }

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

// end of PaneOrganizer class
**Constants Class**

- In our *MovingShapeApp*, we’ve been using absolute numbers in various places
  - Not very extensible! What if we wanted to quickly change the size of our *Scene* or *Shape*?
- The **Constants** class keeps track of a few important numbers
- For our *MovingShapeApp*, the width and height of the *Ellipse* and of the *Panel* it sits in, as well as the start location and move distance

```java
public class Constants {
    // units all in pixels
    public static final double X_RAD = 50;
    public static final double Y_RAD = 50;
    public static final double APP_WIDTH = 200;
    public static final double APP_HEIGHT = 130;
    public static final double BUTTON_SPACING = 30;
    public static final double X_OFFSET = 100;
    public static final double Y_OFFSET = 50;
    public static final double DISTANCE_X = 10;
}
```
public class App extends Application {
    @Override
    public void start(Stage stage) {
        PaneOrganizer organizer = new PaneOrganizer();
        Scene scene = new Scene(organizer.getRoot(),
            Constants.APP_WIDTH, Constants.APP_HEIGHT);
        stage.setScene(scene);
        stage.setTitle("MovingShape!");
        stage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}

public class PaneOrganizer {
    private BorderPane _root;
    private Ellipse _ellipse;

    public PaneOrganizer() {
        _root = new BorderPane();
        Pane shapePane = new Pane();
        _ellipse = new Ellipse(Constants.X_RAD, Constants.Y_RAD);
        shapePane.getChildren().add(_ellipse);
        _root.setCenter(shapePane);
        this.setupShape();
        this.setupButtons();
    }
    public Pane getRoot() {
        return _root;
    }
    private void setupShape() {
        _ellipse.setFill(Color.RED);
        _ellipse.setCenterX(Constants.X_OFFSET);
        _ellipse.setCenterY(Constants.Y_OFFSET);
    }
    private void setupButtons() {
        HBox buttonPane = new HBox();
        _root.setBottom(buttonPane);
        Button b1 = new Button("Move Left!");
        Button b2 = new Button("Move Right!");
        buttonPane.getChildren().addAll(b1, b2);
        buttonPane.setSpacing(Constants.BUTTON_SPACING);
        buttonPane.setAlignment(Pos.CENTER);
        b1.setOnAction(new MoveHandler(true));
        b2.setOnAction(new MoveHandler(false));
    }
    private class MoveHandler implements EventHandler<ActionEvent> {
        private int _distance;
        public MoveHandler(boolean isLeft) {
            _distance = Constants.DISTANCE_X;
            if (isLeft) {
                _distance *= -1;
            }
        }
        public void handle(ActionEvent event) {
            _ellipse.setCenterX(_ellipse.getCenterX() + _distance);
        }
    }
} // end of private MoveHandler class
} // end of PaneOrganizer class
Creating Composite Shapes

• What if we want to display something more elaborate than a single, simple geometric primitive?
• We can make a composite shape by combining two or more shapes!
Spec: **MovingAlien**

- Transform **MovingShape** into **MovingAlien**
- An alien should be displayed on the central **Pane**, and should be moved back and forth by **Buttons**
MovingAlien: Design

- Create a class, `Alien` to model a composite shape
- Define composite shape’s capabilities in `Alien` class
- Give `Alien` a `setLocation()` method that positions each component (face, left eye, right eye, all `Ellipses`)
  - Example of delegation
Turning *MovingShape* into *MovingAlien*

1. **Create** *Alien* **class** to model composite shape, and add each component of *Alien* to *alienPane’s* list of children

2. Be sure to explicitly define any methods that we need to call on *Alien* from within *PaneOrganizer*, such as *location setter/getter methods*!

3. **Modify** *PaneOrganizer* to contain an *Alien* instead of an *Ellipse*
Alien Class

- The Alien class is our composite shape
- It contains three Ellipses—one for the face and one for each eye
- Constructor instantiates these Ellipses, sets their initial sizes/colors, and adds them as children of the alienPane—which was passed in as a parameter

```java
public class Alien {
    private Ellipse _face;
    private Ellipse _leftEye;
    private Ellipse _rightEye;

    public Alien(Pane alienPane) {
        _face = new Ellipse(90, 50);
        _face.setFill(Color.CHARTREUSE);
        _leftEye = new Ellipse(30, 50);
        _leftEye.setFill(Color.BLACK);
        _rightEye = new Ellipse(150, 50);
        _rightEye.setFill(Color.BLACK);
        alienPane.getChildren().addAll(_face, _leftEye, _rightEye);
    }

    public void setXLoc(double x) {
        _face.setCenterX(x);
        _leftEye.setCenterX(x - 30);
        _rightEye.setCenterX(x + 30);
    }

    public double getXLoc() {
        return _face.getCenterX();
    }
}
```
Turning **MovingShape** into **MovingAlien**

1. Create **Alien** class to model composite shape, and add each component of **Alien** to **alienPane**’s list of children

2. Be sure to explicitly define any methods that we need to call on **Alien** from within **PaneOrganizer**, such as *location setter/getter methods*!

3. Modify **PaneOrganizer** to contain an **Alien** instead of an **Ellipse**
Alien Class

- In MovingShapeApp, the following call is made from within our MoveHandler's handle method in order to move the Ellipse:

  ```java
  _ellipse.setCenterX(_ellipse.getCenterX() + _distance);
  ```

- Because we call `getCenterX()` and `setCenterX(...)` on our shape from within the PaneOrganizer class, we must define methods such as `setLocX(...)` and `getLocX()` to set the Alien's location in the Alien class!

- This allows our Alien class to function like an Ellipse in our program!

- Note: most of the time when you are creating complex shapes, you will want to define a more extensive `setLocation(double x, double y)` method rather than having a separate method for the X or Y location
**Alien Class**

A. Define Alien’s setXLoc(...) by setting center X of face, left and right eyes (same for setYLoc)

- Note: relative positions between the Ellipses remains the same

```java
public class Alien {
    private Ellipse _face;
    private Ellipse _leftEye;
    private Ellipse _rightEye;

    public Alien(Pane alienPane) {
        _face = new Ellipse(0, 0);
        _face.setFill(Color.CHARTREUSE);
        _leftEye = new Ellipse(0, 0);
        _leftEye.setFill(Color.BLACK);
        _rightEye = new Ellipse(0, 0);
        _rightEye.setFill(Color.BLACK);

        alienPane.getChildren().addAll(_face, _leftEye, _rightEye);
    }

    public void setXLoc(double x) {
        _face.setCenterX(x);
        _leftEye.setCenterX(x - Constants.EYE_OFFSET);
        _rightEye.setCenterX(x + Constants.EYE_OFFSET);
    }
```

54/66
**Alien Class**

A. Define **Alien’s setXLoc(...)** by setting center X of face, left and right eyes (same for setYLoc)
   - Note: relative positions between the Ellipse’s remains the same

B. Define **getXLoc() method:** the horizontal center of the Alien will always be center of face Ellipse
Alien Class

A. Define Alien’s `setXLoc(...)` by setting center X of face, left and right eyes (same for `setYLoc`)
   - note: relative positions between the ellipses remains the same

B. Define `getXLoc()` method: the horizontal center of the Alien will always be center of face Ellipse

C. Set starting X location of Alien in constructor!
Turning **MovingShape** into **MovingAlien**

1. Create *Alien* class to model composite shape, and add each component of *Alien* to *alienPane*’s list of children

2. Be sure to explicitly define any methods that we need to call on *Alien* from within *PaneOrganizer*, such as *location setter/getter methods*!

3. **Modify** *PaneOrganizer* to contain an *Alien* instead of an *Ellipse*
**PaneOrganizer Class**

- Only have to make a few changes to PaneOrganizer!
- Instead of knowing about an Ellipse called _ellipse, knows about an Alien called _alien
- Change the shapePane to be an alienPane (we could have called it anything!)

```java
public class PaneOrganizer {
    private BorderPane _root;
    private Alien _alien;

    public PaneOrganizer() {
        _root = new BorderPane();
        Pane alienPane = new Pane();
        _alien = new Alien(alienPane);
        _root.setCenter(alienPane);
        this.setupShape();
        this.setupButtons();
    }

    public Pane getRoot() {
        return _root;
    }

    private void setupShape() {
        _ellipse.setFill(Color.RED);
        _ellipse.setCenterX(ChipConstants.X_OFFSET);
        _ellipse.setCenterY(ChipConstants.Y_OFFSET);
    }

    private void setupButtons() {
        Hbox buttonPane = new Hbox();
        _root.setBottom(buttonPane);
        Button b1 = new Button("Move Left");
        Button b2 = new Button("Move Right");
        buttonPane.getChildren().addAll(b1, b2);
        buttonPane.spacing(30);
        b1.setOnAction(new MoveHandler(true));
        b2.setOnAction(new MoveHandler(false));
    }

    /* private class MoveHandler elided */
}
```
PaneOrganizer Class

- `setupShape()` method is no longer needed, as we now setup the Alien within the Alien class
PaneOrganizer Class

• setupShape() method is no longer needed, as we now setup the Alien within the Alien class
  o remember that we set a default location for the Alien in its constructor:

    this.setXLoc(Commands.START_X_OFFSET);
**PaneOrganizer Class**

- Last modification we have to make is from within the `MoveHandler` class, where we will swap in `_alien` for `_ellipse` references.

- We implemented `setXLoc(...)` and `getXLoc()` methods in `Alien` so `MoveHandler` can call them.
public class App extends Application {
    @Override
    public void start(Stage stage) {
        PaneOrganizer organizer = new PaneOrganizer();
        Scene scene = new Scene(organizer.getRoot(),
            Constants.APP_WIDTH, Constants.APP_HEIGHT);
        stage.setScene(scene);
        stage.setTitle("MovingAlien!");
        stage.show();
    }

global void main(String[] args) {
    launch(args);
}
}

public class Alien {
    private Ellipse _face;
    private Ellipse _leftEye;
    private Ellipse _rightEye;
    public Alien(Pane root) {
        _face = new Ellipse(Constants.X_RAD, Constants.Y_RAD);
        _face.setFill(Color.CHARTREUSE);
        _leftEye = new Ellipse(Constants.EYE_X, Constants.EYE_Y);
        _rightEye = new Ellipse(Constants.EYE_X, Constants.EYE_Y);
        root.getChildren().addAll(_face, _leftEye, _rightEye);
    }

    public void setXLoc(double x) {
        _face.setCenterX(x);
        _leftEye.setCenterX(x - Constants.EYE_OFFSET);
        _rightEye.setCenterX(x + Constants.EYE_OFFSET);
    }

    public double getXLoc() {
        return _face.getCenterX();
    }
}

public class PaneOrganizer {
    private BorderPane _root;
    private Alien _alien;
    public PaneOrganizer() {
        _root = new BorderPane();
        Pane alienPane = new Pane();
        _alien = new Alien(alienPane);
        _root.setCenter(alienPane);
        this.setupButtons();
    }

    public Pane getRoot() {
        return _root;
    }

    private void setupButtons() {
        Hbox buttonPane = new Hbox();
        _root.setBottom(buttonPane);
        Button b1 = new Button("Move Left");
        Button b2 = new Button("Move Right");
        buttonPane.getChildren().addAll(b1, b2);
        buttonPane.setSpacing(30);
        b1.setOnAction(new MoveHandler(true));
        b2.setOnAction(new MoveHandler(false));
    }

    private class MoveHandler implements EventHandler<ActionEvent> {
        private int _distance;
        public MoveHandler(boolean isLeft) {
            _distance = Constants.DISTANCE_X;
            if (isLeft) {
                _distance *= -1;
            }
        }

        public void handle(ActionEvent event) {
            _alien.setXLoc(_alien.getXLoc() + _distance);
        }
    }
}
Your Next Project: Cartoon!

• You’ll be building a JavaFX application that displays your own custom “cartoon”, much like the examples in this lecture
• But your cartoon will be animated!
Your Next Project: Cartoon!

• How can we animate our cartoon (e.g. make the cartoon move across the screen)?
• As in film and video animation, 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 smoothly changing size, orientation, shape, etc.
Animation in Cartoon

• Use a TimeLine to create incremental change
• It’ll be up to you to figure out the details… but for each repetition of the KeyFrame, your cartoon should move a small amount!
Announcements

• Cartoon and Cartoon DQs have been released
  o Cartoon DQs are due this Friday 10/16 at 10pm
  o Read the handouts carefully
  o Read the handouts carefully
  o Read the handouts carefully

• Review the Graphics lectures, the Shapes Documentation, and the JavaFX Guide

• Lab 4 has been released