Lecture 10
Graphics Part III – Building up to Cartoon
Outline

• **Shapes**
  o example: *MovingShape*
  o *App, PaneOrganizer, and MoveHandler* classes

• **Constants**
  o Lecture Question: Slide 41

• **Composite Shapes**
  o example: *Alien*
  o Lecture Question: Slide 53, Slide 61

• **Cartoon**
Example: **MovingShape**

• Specification: App that displays a shape and buttons that shift position of the shape left and right by a fixed increment

• Purpose: Practice working with absolute positioning of Panes, various Shapes, and more event handling!
Process: MovingShapeApp

1. Write an 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 a ShapeMover class which will be responsible for shape movement and other logic. It is instantiated in the PaneOrganizer’s constructor

4. Write setupShape() and setupButtons() helper methods to be called within ShapeMover’s constructor. These will factor out code for modifying our sub-Panes

5. Register Buttons with EventHandlers that handle Buttons’ ActionEvents (clicks) by moving Shape correspondingly, within the ShapeMover class
MovingShapeApp: App Class (1/3)

*NOTE: Exactly the same process as previous examples*

1a. 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();
    }
}
```
MovingShapeApp: App Class (2/3)

*NOTE: Exactly the same process as previous examples*

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 (in this case 200x200)

```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();
    }
}
```
MovingShapeApp: App Class (3/3)

*NOTE: Exactly the same process as previous examples*

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 (in this case 200x200)

1c. 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("MovingShape");
        stage.show();
    }
}
```
Process: **MovingShapeApp**

1. Write an `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 all necessary Panes and initialize the `ShapeMover` class

3. Write a `ShapeMover` class which will be responsible for shapes creation, movement, and other logic. It is instantiated in the `PaneOrganizer`’s constructor

4. Write `setupShape()` and `setupButtons()` helper methods to be called within `ShapeMover`’s constructor. These will factor out code for modifying our sub-Panes

5. Register `Buttons` with `EventHandlers` that handle `Buttons`’ `ActionEvents` (clicks) by moving `Shape` correspondingly, within the `ShapeMover` class
MovingShapeApp: PaneOrganizer
Class (1/3)

2a. Instantiate the root Pane and store it in the instance variable _root

```
public class PaneOrganizer {
    private Pane _root;

    public PaneOrganizer() {
        _root = new Pane();
    }
}
```
MovingShapeApp: PaneOrganizer Class (2/3)

2a. Instantiate the root Pane and store it in the instance variable _root

2b. 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;
    }
}
```
MovingShapeApp: PaneOrganizer Class (3/3)

2a. Instantiate the root Pane and store it in the instance variable _root

2b. Create a public getRoot() method that returns _root

2c. Create a new instance of ShapeMover(), defined next. Pass _root as argument (The constructor of ShapeMover() takes in a Pane Slide 13)

```java
public class PaneOrganizer {
    private Pane _root;
    public PaneOrganizer() {
        _root = new Pane();
        new ShapeMover(_root);
    }
    public Pane getRoot() {
        return _root;
    }
}
```
Process: **MovingShapeApp**

1. Write an `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 a `ShapeMover` class which will be responsible for shape movement and other logic. It is instantiated in the `PaneOrganizer`’s constructor

4. Write `setupShape()` and `setupButtons()` helper methods to be called within `ShapeMover`’s constructor. These will factor out code for modifying our sub-Panes

5. Register `Buttons` with `EventHandlers` that handle `Buttons`’s `ActionEvents` (clicks) by moving `Shape` correspondingly, within the `ShapeMover` class
• **PaneOrganizer** was getting too complex: Factor out the program logic into **ShapeMover**; it will
  - set up the shape graphically and logically
  - set up the buttons graphically and logically
  - set up the **EventHandler** and link it to the buttons

3a. Make the constructor of **ShapeMover** take in the root **Pane** (created in **PaneOrganizer**, see slide 11)

```java
public class ShapeMover {
    private Ellipse _ellipse;

    public ShapeMover(Pane root) {
        // constructor implementation
    }
}
```
3a. Make the constructor of ShapeMover take in the root Pane

3b. Create an instance variable _ellipse and initialize an ellipse
3a. Make the constructor of ShapeMover take in the root Pane

3b. Create an instance variable _ellipse and initialize an ellipse

3c. Add the ellipse as a child of the root Pane

```java
public class ShapeMover {
    private Ellipse _ellipse;

    public ShapeMover(Pane root) {
        _ellipse = new Ellipse(50, 50);
        root.getChildren().add(_ellipse);
    }
}
```
3a. Make the constructor of `ShapeMover` take in the `root` `Pane`

3b. Create an instance variable `_ellipse` and initialize an `ellipse`

3c. Add the ellipse as a child of the `root` `Pane`

3d. Call `setupShape()` and `setupButtons()`, defined next

```java
public class ShapeMover {
    private Ellipse _ellipse;

    public ShapeMover(Pane root) {
        _ellipse = new Ellipse(50, 50);
        root.getChildren().add(_ellipse);

        this.setupShape();
        this.setupButtons(root);
    }
}
```
Process: **MovingShapeApp**

1. Write an `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 a `ShapeMover` class which will be responsible for shape movement and other logic. It is instantiated in the `PaneOrganizer`’s constructor

4. Write `setupShape()` and `setupButtons()` helper methods to be called within `ShapeMover`’s constructor. These will factor out code for modifying our sub-Panes

5. Register `Buttons` with `EventHandlers` that handle `Buttons’ ActionEvents` (clicks) by moving `Shape` correspondingly, within the `ShapeMover` class
Aside: helper methods

• 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 ShapeMover constructor—it is good practice to factor out code into **helper methods** that are called within the constructor—another use of the delegation pattern
  o **setupShape()** fills and positions *Ellipse*
  o **setupButtons()** adds and positions *Buttons*, and registers them with their appropriate *EventHandler*

• Helper methods of the form **setupX()** are fancy initializing assignments. Should be used to initialize variables, but **not** for arbitrary/non-initializing code.

• Generally, helper methods should be **private** – more on this in a moment
MovingShapeApp: 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?
  o only ShapeMover() should be allowed to initialize the color and location of the Ellipse
  o private methods are not directly inherited and are not accessible to subclasses—though inherited superclass methods may make use of them w/o the subclass knowing about them!

```java
public class ShapeMover {
    private Ellipse _ellipse;

    public ShapeMover(Pane root) {
        _ellipse = new Ellipse(50, 50);
        root.getChildren().add(_ellipse);
        this.setupShape();
        this.setupButtons(root);
    }

    private void setupShape() {
        _ellipse.setFill(Color.RED);
        _ellipse.setCenterX(50);
        _ellipse.setCenterY(50);
    }
}
```
Aside: **PaneOrganizer** Class (1/3)

- 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!

```java
left.relocate(50,165);
right.relocate(120,165);
left.relocate(100,180);
right.relocate(150,180);
left.relocate(50,150);
right.relocate(120,150);
left.relocate(50,165);
right.relocate(120,165);
```

Is there a better way? ... **hint**: leverage Scene Graph hierarchy and delegation!
Aside: PaneOrganizer Class (2/3)

• 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
  o add Buttons to HBox, to align horizontally

• Continuing to improve our design, use a BorderPane as root to use its layout manager

• Now need to add Ellipse to the root
  o could simply add Ellipse to CENTER of root BorderPane
  o but this won’t work—if BorderPane dictates placement of Ellipse we won’t be able to update its position with Buttons
  o instead: create a Pane to contain Ellipse and add the Pane as child of root! Can adjust Ellipse within its shapePane independently!
Aside: PaneOrganizer Class (3/3)

• This makes use of the built-in layout capabilities available to us in JavaFX!
• Also makes symmetry between the panel holding a shape (in Cartoon, this panel will hold composite shapes that you’ll make) and the panel holding our buttons
• Note: this is only one of many design choices for this application!
  o keep in mind all of the different layout options when designing your programs!
  o using absolute positioning for entire program is most likely not best solution—where possible, leverage power of layout managers (BorderPane, HBox, VBox,...)
MovingShapeApp: update to BorderPane

4a. Change root to a BorderPane

public class PaneOrganizer {
    private BorderPane _root;

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

        new ShapeMover(_root);
    }

    public Pane getRoot() {
        return _root;
    }
}
MovingShapeApp: update to BorderPane

4a. Change root to a BorderPane

4b. Create a Pane to contain Ellipse. To add shapePane to center of BorderPane, call setCenter(shapePane) on root

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

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

        // setup shape pane
        Pane shapePane = new Pane();
        _root.setCenter(shapePane);

        new ShapeMover(_root);
    }

    public Pane getRoot() {
        return _root;
    }
}
```
4c. Instantiate a new HBox, then add it as child of BorderPane, in bottom position

```java
class PaneOrganizer {
    private BorderPane _root;

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

        // setup shape pane
        Pane shapePane = new Pane();
        _root.setCenter(shapePane);

        HBox buttonPane = new HBox();
        _root.setBottom(buttonPane);

        new ShapeMover(_root);
    }

    public Pane getRoot() {
        return _root;
    }
}
```
4c. Instantiate a new **HBox**, then add it as child of **BorderPane**, in bottom position

4d. Modify the argument of **ShapeMover** to take in the **shapePane** and the **buttonPane** instead of the root **Pane**
MovingShapeApp: Ellipse in the shapePane

4e. In the ShapeMover class, add the ellipse as a child of the shapePane instead of root

- note: none of the code in our setupShape() method needs to be updated since it accesses _ellipse directly... with this redesign, _ellipse now is just **graphically** contained within a different Pane (the shapePane) and now in the center of the root because we called setCenter(shapePane)

- and ShapeMover can still access the ellipse because it remains its instance variable!
  - this could be useful if we want to change any properties of the Ellipse later on, e.g., updating its x and y position, or changing its color
  - illustration of graphical vs. logical containment

```java
public class ShapeMover {
    private Ellipse _ellipse;

    public ShapeMover(Pane shapePane, HBox buttonPane) {
        _ellipse = new Ellipse(50, 50);
        shapePane.getChildren().add(_ellipse);
        this.setupShape();
        this.setupButtons(buttonPane);
    }

    /* setupShape elided! This method sets the color and * initial position of the ellipse */
```
MovingShapeApp: setupButtons() method (1/4)

4f. In the ShapeMover class, create a method called setupButtons() which takes in the buttonPane and instantiate two Buttons.

```java
public class ShapeMover {
    private Ellipse _ellipse;

    public ShapeMover(Pane shapePane, HBox buttonPane) {
        _ellipse = new Ellipse(50, 50);
        shapePane.getChildren().add(_ellipse);

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

    // setupShape elided!

    private void setupButtons(HBox buttonPane) {
        Button b1 = new Button("move left");
        Button b2 = new Button("move right");
    }
}
```
4f. In the `ShapeMover` class, create a method called `setupButtons()` which takes in the `buttonPane` and instantiate two Buttons.

4g. Add the Buttons as children of the new `HBox`.

   - order matters when adding children to `Panes`. For this `HBox`, `b1` will be to the left of `b2` because it is added first in the list of arguments in `addAll(...)`.

```java
public class ShapeMover {
    private Ellipse _ellipse;

    public ShapeMover(Pane shapePane, HBox buttonPane) {
        _ellipse = new Ellipse(50, 50);
        shapePane.getChildren().add(_ellipse);

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

    // setupShape elided!

    private void setupButtons(HBox buttonPane) {
        Button b1 = new Button("move left");
        Button b2 = new Button("move right");
        buttonPane.getChildren().addAll(b1, b2);
    }
}
```
4h. Set horizontal spacing between Buttons as you like

```java
public class ShapeMover {
    private Ellipse _ellipse;

    public ShapeMover(Pane shapePane, HBox buttonPane) {
        _ellipse = new Ellipse(50, 50);
        shapePane.getChildren().add(_ellipse);

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

    // setupShape elided!

    private void setupButtons(HBox buttonPane) {
        Button b1 = new Button("move left");
        Button b2 = new Button("move right");
        buttonPane.getChildren().addAll(b1, b2);

        buttonPane.setSpacing(30);
    }

```
MovingShapeApp: setupButtons() method (4/4)

4h. Set horizontal spacing between Buttons as you like

4i. We will come back to the ShapeMover class in the next step in order to register Buttons with their EventHandlers, but first we should define the EventHandler

```java
public class ShapeMover {
    private Ellipse _ellipse;
    public ShapeMover(Pane shapePane, HBox buttonPane) {
        _ellipse = new Ellipse(50, 50);
        shapePane.getChildren().add(_ellipse);
        this.setupShape();
        this.setupButtons(buttonPane);
    }

    // setupShape elided!

    private void setupButtons(HBox buttonPane) {
        Button b1 = new Button("move left");
        Button b2 = new Button("move right");
        buttonPane.getChildren().addAll(b1, b2);
        buttonPane.setSpacing(30);
    }
}
```
Process: **MovingShapeApp**

1. Write an **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 a `ShapeMover` class which will be responsible for shape movement and other logic. It is instantiated in `PaneOrganizer`’s constructor

4. Write `setupShape()` and `setupButtons()` helper methods to be called within `ShapeMover`’s constructor. These will factor out code for modifying our sub-Panes

5. **Register Buttons with EventHandlers that handle Buttons’ ActionEvents (clicks) by moving Shape correspondingly, within the ShapeMover class**
Aside: 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 that will have two instances
  - specifics determined by parameters passed into the constructor!
  - admittedly, this is not an obvious design—these kinds of simplifications typically have to be learned…
5a. Declare an instance variable _distance that will be initialized differently depending on whether the isLeft argument is true or false

```java
public class ShapeMover {
    // other code elided
    public ShapeMover(Pane shapePane, HBox buttonPane) {
        // other code elided
    }

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

        public MoveHandler(boolean isLeft) {
            // other code elided
        }

        public void handle(ActionEvent e) {
            // other code elided
        }
    }
}
```

NOTE: such methods returning a boolean are called predicates
MovingShapeApp: MoveHandler (2/3)

5a. Declare an instance variable _distance that will be initialized differently depending on whether the isLeft argument is true or false

5b. 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 ShapeMover {
    // other code elided

    public ShapeMover(Pane shapePane, Hbox buttonPane) {
        // other code elided
    }

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

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

        public void handle(ActionEvent e) {
            // other code elided
        }
    }
}
```
MovingShapeApp: MoveHandler (3/3)

5a. Declare an instance variable _distance that will be initialized differently depending on whether the isLeft argument is true or false

5b. Set _distance to 10 initially – if the registered Button isLeft, change _distance to -10 so the Ellipse moves in the opposite direction

5c. Implement the handle method to move the Ellipse by _distance in the horizontal direction

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

    public ShapeMover(Pane shapePane, HBox buttonPane) {
        // other code elided
    }

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

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

        public void handle(ActionEvent e) {
            //called by JFX
            _ellipse.setCenterX(_ellipse.getCenterX()+_distance);
        }
    }

    // other code elided
}
```
MovingShapeApp: back to setupButtons()

Register Buttons with their EventHandlers by calling setOnAction() and passing in our instances of MoveHandler, which we just created!

```java
public class ShapeMover {
  private Ellipse _ellipse;

  public ShapeMover(Pane shapePane, HBox buttonPane) {
    _root = new BorderPane();
    // setup of shape elided!
    this.setupButtons(buttonPane);
  }
  // setupShape elided!
  private void setupButtons(Hbox 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));
  }
  // MoveHandler elided!
}
```

This is where we set isLeft
Logical Containment Diagram

- Note this is quite different from the Scene Graph, which only handles graphical containment
- **PaneOrganizer** contains three Panes (**root**, **shapePane**, **buttonPane**) and the **ShapeMover**
  - Notice **PaneOrganizer** Delegates the handling of graphical shapes to **ShapeMover**
- **ShapeMover** contains an **Ellipse**, **Buttons**, and an **EventHandler** (**MoveHandler**)
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.scene.layout.*;
import javafx.scene.shape.Ellipse;
// imports for the ShapeMover class
import javafx.scene.paint.Color;
import javafx.event.*;
import javafx.geometry.Pos;
import javafx.scene.control.Button;
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);
  }
}
public class PaneOrganizer {
  private BorderPane _root;

  public PaneOrganizer() {
    _root = new BorderPane();
    Pane shapePane = new Pane();
    _root.setCenter(shapePane);
    HBox buttonPane = new HBox();
    _root.setBottom(buttonPane);
    new ShapeMover(shapePane, buttonPane);
  }
  public PanegetRoot() {
    return _root;
  }
}
public class ShapeMover {
  private Ellipse _ellipse;

  public ShapeMover(Pane shapePane, HBox buttonPane) {
    _ellipse = new Ellipse(50, 50);
    shapePane.getChildren().add(_ellipse);
    this.setupShape();
    this.setupButtons(buttonPane);
  }

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

  private void setupButtons(HBox 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 double _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
} // end of ShapeMover class
The Whole App

public class ShapeMover {
  private Ellipse _ellipse;

  public ShapeMover(Pane shapePane, HBox buttonPane) {
    _ellipse = new Ellipse(50, 50);
    shapePane.getChildren().add(_ellipse);
    this.setupShape();
    this.setupButtons(buttonPane);
  }

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

  private void setupButtons(HBox 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 double _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
} // end of ShapeMover class
Reminder: *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* to improve compile time?
- Our *Constants* class will keep track of a few important numbers
- For our *MovingShapeApp*, make constants for width and height of the *Ellipse* and of the *Pane* it sits in, as well as the start location and distance moved

```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;
    /* X_OFFSET is the graphical offset from the edge of the screen to where we want the X value of the Ellipse */
    public static final double X_OFFSET = 100;
    public static final double Y_OFFSET = 50;
    public static final double DISTANCE_X = 10;
}
```
Lecture Question

When should you define a value in a Constants class?

A. When you use the value in more than one place.
B. Whenever the value will not change throughout the course of the program.
C. When the value is nontrivial (i.e., not 0 or 1)
D. All of the above.
public class ShapeMover {
    private Ellipse _ellipse;
    public ShapeMover(Pane shapePane, Hbox buttonPane) {
        _ellipse = new Ellipse(Constants.X_RAD, Constants.Y_RAD);
        shapePane.getChildren().add(_ellipse);
        this.setupShape();
        this.setupButtons(buttonPane);
    }

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

    private void setupButtons(Hbox buttonPane) {
        Button b1 = new Button("move left");
        Button b2 = new Button("move right");
        buttonPane.getChildren().addAll(b1, b2);
        buttonPane.setSpacing(Constants.BUTTON_SPACING);
        b1.setOnAction(new MoveHandler(true));
        b2.setOnAction(new MoveHandler(false));
    }

    private class MoveHandler implements EventHandler<ActionEvent> {
        private double _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 ShapeMover class

The Whole App

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;
    public PaneOrganizer() {
        _root = new BorderPane();
        Pane shapePane = new Pane();
        _root.setCenter(shapePane);
        HBox buttonPane = new HBox();
        _root.setBottom(buttonPane);
        new ShapeMover(shapePane, buttonPane);
    }

    public Pane getRoot() {
        return _root;
    }
}
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!
Specifications: 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)
  o another example of delegation pattern
Process: 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 *AlienMover* (which used to be *ShapeMover*), such as *location* setter/getter methods!

3. Modify *AlienMover* 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
• Although Alien class deals with each component of the composite shape individually, every component should reside on the same pane as all other components
  o thus, must pass pane as a parameter to allow Alien class to define methods for manipulating composite shape in pane

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

    public Alien( Pane alienPane ) { //Alien lives in passed Pane
        _face = new Ellipse(Constants.X_RAD, Constants.Y_RAD);
        _face.setFill(Color.CHARTREUSE);
        /*EYE_X and EYE_Y are constants referring to the width and height of the eyes, the eyes' location/center is changed later in the program.*/
        _leftEye = new Ellipse(Constants.EYE_X, Constants.EYE_Y);
        _leftEye.setFill(Color.BLACK);
        _rightEye = new Ellipse(Constants.EYE_X, Constants.EYE_Y);
        _rightEye.setFill(Color.BLACK);
        alienPane.getChildren().addAll(_face, _leftEye, _rightEye);
        this.setXLoc(Constants.START_X_OFFSET);
    }

    public void setXLoc(double x) {
        _face.setCenterX(x);
    }

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

Note: Order matters when you add children to a Pane! The arguments are added in that order graphically and if there is overlap, the shape later in the parameter list will lie wholly or partially on top of the earlier one. For this example, _face is added first, then _leftEye and _rightEye on top. The inverse order would be wrong!
Process: 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 AlienMover (which used to be ShapeMover), such as location setter/getter methods!

3. Modify AlienMover 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 called JavaFX’s `getCenterX()` and `setCenterX(...)` on our shape from within the ShapeMover class, we must now define our own equivalent 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
MovingAlien: Alien Class (1/3)

2a. Define Alien’s setXLoc(...) by setting center X of face, left and right eyes (same for setYLoc); note use of additional constants

- 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(Constants.X_RAD, Constants.Y_RAD);
        _face.setFill(Color.CHARTREUSE);
        _leftEye = new Ellipse(Constants.EYE_X, Constants.EYE_Y);
        _leftEye.setFill(Color.BLACK);
        _rightEye = new Ellipse(Constants.EYE_X, Constants.EYE_Y);
        _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);
    }
}
```
2a. Define Alien’s `setXLoc(...)` by using ellipse’s `setCenter()`... on face, left and right eyes (same for `setYLoc()`);
   o note: relative positions between the Ellipses remains the same

2b. Define `getXLoc()` method: the horizontal center of the Alien will always be center of _face Ellipse

```java
class Alien {
    private Ellipse _face;
    private Ellipse _leftEye;
    private Ellipse _rightEye;
    public Alien(Pane alienPane) {
        _face = new Ellipse(Constants.X_RAD, Constants.Y_RAD);
        _face.setFill(Color.CHARTREUSE);
        _leftEye = new Ellipse(Constants.EYE_X, Constants.EYE_Y);
        _leftEye.setFill(Color.BLACK);
        _rightEye = new Ellipse(Constants.EYE_X, Constants.EYE_Y);
        _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);
    }
    public double getXLoc() {
        return _face.getCenterX();
    }
}
```
MovingAlien: Alien Class (3/3)

2a. 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

2b. Define `getXLoc()` method: the horizontal center of the Alien will always be center of _face Ellipse

2c. Set starting X location of Alien in constructor!
Lecture Question

Which **House** constructor makes the correct composite shape, given the rest of the program is set up correctly?

A. 

```java
public House (Pane housePane) {
    _foundation = new Rectangle( Constants.X, Constants.Y);
    _window = new Rectangle( Constants.WIND_X, Constants.WIND_Y);
    _door = new Rectangle( Constants.DOOR_X, Constants.DOOR_Y);
    //code to fill _foundation, _window, _door elided
    housePane.getChildren().addAll(_foundation, _window, _door);
    this.setXLoc( Constants.INITIAL_X_OFFSET);
}
```

B. 

```java
public House () {
    _foundation = new Rectangle( Constants.X, Constants.Y);
    _window = new Rectangle( Constants.WIND_X, Constants.WIND_Y);
    _door = new Rectangle( Constants.DOOR_X, Constants.DOOR_Y);
    //code to fill _foundation, _window, _door elided
    new Pane().getChildren().addAll(_foundation, _window, _door);
    new Pane().setX( Constants.INITIAL_X_OFFSET);
}
```

C. 

```java
public House (Pane housePane) {
    _foundation = new Rectangle();
    _window = new Rectangle();
    _door = new Rectangle();
    //code to fill _foundation, _window, _door elided
    housePane.getChildren().addAll(_foundation, _window, _door);
    this.setXLoc( Constants.INITIAL_X_OFFSET);
}
```

D. 

```java
public House (Pane housePane) {
    _foundation = new Rectangle( Constants.X, Constants.Y);
    _window = new Rectangle( Constants.WIND_X, Constants.WIND_Y);
    _door = new Rectangle( Constants.DOOR_X, Constants.DOOR_Y);
    //code to fill _foundation, _window, _door elided
    this.setXLoc( Constants.INITIAL_X_OFFSET);
}
```
Process: 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 `AlienMover` (which used to be `ShapeMover`), such as `location` setter/getter methods!

3. **Modify `AlienMover` to contain an `Alien` instead of an `Ellipse`**
MovingAlien: PaneOrganizer Class

• Change the shapePane to be an alienPane (we could have called it anything!)

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

    public PaneOrganizer() {
        _root = new BorderPane();
        Pane alienPane = new Pane();
        _root.setCenter(alienPane);
        HBox buttonPane = new HBox();
        _root.setBottom(buttonPane);
        new AlienMover(alienPane, buttonPane);
    }
    public Pane getRoot() {
        return _root;
    }
}
```
MovingAlien: AlienMover Class (1/3)

• Only have to make a few changes to AlienMover!
• Instead of containing an Ellipse called _ellipse, contains an Alien called _alien
• Change shapePane to be an alienPane (we could have called it anything!)

```java
public class AlienMover {
    private Alien _alien;
    public ShapeMover(Pane alienPane, Hbox buttonPane) {
        _alien = new Alien(alienPane);
        this.setupShape();
        this.setupButtons(buttonPane);
    }
    private void setupShape() {
        _ellipse.setFill(Color.RED);
        _ellipse.setCenterX( Constants.X_OFFSET );
        _ellipse.setCenterY( Constants.Y_OFFSET );
    }
    private void setupButtons(Hbox buttonPane) {
        Button b1 = new Button("Move Left!");
        Button b2 = new Button("Move Right!");
        buttonPane.getChildren().addAll(b1, b2);
        buttonPane.setSpacing( Constants.BUTTON_SPACING );
        b1.setOnAction( new MoveHandler(true));
        b2.setOnAction( new MoveHandler(false));
    }
    // private MoveHandler class elided!
}
```
MovingAlien: AlienMover Class (2/3)

• 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:

```java
this.setXLoc(Constants.START_X_OFFSET);
```

```java
public class AlienMover {
    private Alien _alien;
    public ShapeMover(Pane alienPane, Hbox buttonPane) {
        _alien = new Alien(alienPane);
        this.setupShape();
        this.setupButtons(buttonPane);
    }
    private void setupShape() {
        _ellipse.setFill(Color.RED);
        _ellipse.setCenterX(Constants.X_OFFSET);
        _ellipse.setCenterY(Constants.Y_OFFSET);
    }
    private void setupButtons(Hbox buttonPane) {
        Button b1 = new Button("Move Left!");
        Button b2 = new Button("Move Right!");
        buttonPane.getChildren().addAll(b1, b2);
        buttonPane.setSpacing(Constants.BUTTON_SPACING);
        b1.setOnAction(new MoveHandler(true));
        b2.setOnAction(new MoveHandler(false));
    }
    // private MoveHandler class elided!
}
```
MovingAlien: AlienMover Class (3/3)

• 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

```java
public class AlienMover {
    private Alien _alien;
    public ShapeMover(Pane alienPane, Hbox buttonPane) {
        _alien = new Alien(alienPane);
        this.setupButtons(buttonPane);
    }
    private void setupButtons(Hbox buttonPane) {
        Button b1 = new Button("Move Left!");
        Button b2 = new Button("Move Right!");
        buttonPane.getChildren().addAll(b1, b2);
        buttonPane.setSpacing(Constants.BUTTON_SPACING);
        b1.setOnAction(new MoveHandler(true));
        b2.setOnAction(new MoveHandler(false));
    }
    private class MoveHandler implements EventHandler<ActionEvent> {
        private double _distance;
        public MoveHandler(boolean isLeft) {
            _distance = Constants.DISTANCE_X;
            if (isLeft) {
                _distance *= -1;
            }
        }
        public void handle(ActionEvent event) {
            _alien.setXLoc(_alien.getXLoc() + _distance);
        }
    }
}
```
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();
    }
    public static 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);
        this.setXLoc(Constants.START_X_OFFSET);
        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 AlienMover {
    private Alien _alien;
    public AlienMover(Pane alienPane, Hbox buttonPane) {
        _alien = new Alien(alienPane);
        this.setupButtons(buttonPane);
        private void setupButtons(Hbox buttonPane) {
            Button b1 = new Button("Move Left!");
            Button b2 = new Button("Move Right!");
            buttonPane.getChildren().addAll(b1, b2);
            buttonPane.setSpacing(Constants.BUTTON_SPACING);
            b1.setOnAction(new MoveHandler(true));
            b2.setOnAction(new MoveHandler(false));
        }
    }
    private class MoveHandler implements EventHandler<ActionEvent> {
        private double _distance;
        public MoveHandler(boolean isLeft) {
            _distance = Constants.DISTANCE_X;
            if (isLeft) {
                _distance *= -1;
            }
        }
        public void handle(ActionEvent event) {
            _alien.setXLoc(_alien.getXLoc() + _distance);
        }
    }
}

public class PaneOrganizer {
    private BorderPane _root;
    public PaneOrganizer() {
        _root = new BorderPane();
        Pane alienPane = new Pane();
        _root.setCenter(alienPane);
        HBox buttonPane = new HBox();
        _root.setBottom(buttonPane);
        new AlienMover(alienPane, buttonPane);
    }
    public Pane getRoot() {
        return _root;
    }
}
Delegation: Creating Additional Classes

• Notice how we created another class for our Alien composite shape instead of simply adding each individual shape to PaneOrganizer.

• We also created a class called AlienMover to handle the “game logic”, aka the problem-specific “program logic”, so PaneOrganizer would not be too complicated.

• As your programs get more complex (e.g., two shapes interacting with one another, shapes changing color, etc.), you may want to use this design. Making a separate class for problem-specific logic allows you to avoid complicating PaneOrganizer.

  o Notice that for this example, all of the game logic exists in AlienMover; PaneOrganizer is only responsible for placing Panes and other elements on the screen.

  o this will make PaneOrganizer less cluttered and your whole program much easier to read.

  o keep this in mind for your upcoming and current assignments!!!
Lecture Question

What is the best practice for setting up graphical scenes (according to CS15)?

A. Absolutely position everything using trial and error, and use as few panes as possible.

B. Have any shape be contained in its own pane, and only make classes for composite shapes of more than 5 shapes.

C. Use a top-level class, make classes for more complicated shapes, and store composite shapes, or just generally related objects, within panes.
Your Project: Cartoon! (1/2)

• 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 Project: Cartoon! (2/2)

• 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 (or change in other ways) a small amount!
  - reminder: if we move fast enough and in small enough increments, we get smooth motion!
Announcements

• Cartoon has been released!
  o Early Handin: Monday, 3/1 at 11:59pm
  o On-Time Handin: Wednesday, 3/3 at 11:59pm
  o Late Handin: Friday, 3/5 at 11:59pm

• Top 6 Cartoons win a meal with Andy over zoom

• Cartoon check-ins Thursday-Saturday
  o Meet with your Section TAs to go over your design for Cartoon, if you have not yet!
  o Make sure you have reached the Cartoon checkpoint by your check-in
IT in the News

ft. Socially Responsible Computing!
Robotics

- Definition: a machine that is controlled by computer(s) and can carry out complex actions (semi-) automatically
  - Not all robots are anthropomorphic!

- Types:
  - Pre-programmed (ex. mechanical assembly)
  - Autonomous (ex. Roomba vacuum)
  - Humanoid (ex. Sophia)
  - Tele-operated (ex. unmanned aeronautic vehicles (UAVs) aka drones)
  - Augmenting (ex. prosthetic limbs)
UAVs: Positive use cases (medicine + supplies)

- **Humanitarian/medical aid**
  - **Rwanda**: UAVs (by drone startup Zipline) transport supplies in far less time than ground travel
    - example: blood transfusions take 6 minutes to deliver via drone, vs. 3 hours by car (>96% less time!)

- **COVID-19 vaccine distribution**
  - **February 2021**: Zipline announces plans to deliver all leading COVID vaccines in US, Rwanda, China, Nigeria, and more
    - could help those living in rural regions, “pharmacy deserts”, to access vaccines more quickly!

*source: TechCrunch, 2020*
UAVs: Positive use cases (documentation + activism)

- **2016-2017**: drones used by protesters at Standing Rock, ND
  - revealed proximity of pipeline to tribe’s water source
  - recorded police brutality + provided counter-evidence to law enforcement’s false narratives

- **May 2020**: photographer George Steinmetz exposed NYC COVID “mass grave”
  - Hart Island = where city buried COVID victims whose bodies were not claimed for private burial
    - graves dug by Rikers Island jail inmates
  - NYPD confiscated drone + detained Steinmetz, resulted in legal battle
UAVs: Negative use cases (CW: policing, war/conflict)

- **Surveillance**
  - Drones used for public **and private** surveillance, combined with other tech (facial recognition, terrain mapping technologies, cell signal trackers)
    - ex. FBI used drones in 2015 to surveil protestors in Baltimore

- **Warfare**
  - UAVs have been used by US military since ~2001 to surveil and perform unmanned strikes
  - Argument: use of drones saves lives of US soldiers
    - **but** distance also enables error, dehumanization of targets, “gamification” of warfare
    - up to 25% of victims are civilians, ~25% of civilian victims are children

Infrared camera footage of Freddie Gray protests, captured by FBI drone

source: ACLU
Robotics @ Brown!

- Faculty:
  - Stefanie Tellex
  - George Konidaris
  - Michael Littman

- Groups:
  - Humanity Centered Robotics Initiative
  - Humans 2 Robots Lab
  - Intelligent Robot Lab
  - RLAB (reinforcement learning & adaptive behavior) group

Learn more @ robotics.cs.brown.edu