CS Diversity Student Advocates

As student advocates, our goal is to improve how our department handles academic and social diversity issues. We run a series of diversity initiatives throughout the year, and are available for office hours by appointment.

Contact us:

diversity.advocates@lists.cs.brown.edu

Inclusivity Feeback Form:

http://tinyurl.com/CSInclusivityForm

Diagnostic Question 1

```
public class CS15TAs {
    private HeadTA _helen;
    private HeadTA _jeff;

    public CS15TAs() {
       HeadTA catherine = new HeadTA();
       HeadTA amos = new HeadTA();
       _helen = new HeadTA();
       _jeff = new HeadTA();
    }

}
```

Which of the following is the correct classification of local and instance variables in the `CS15TAs` class?

A. `catherine` and `amos` are instance variables, `_helen` and `_jeff` are local variables

B. All four are instance variables

C. All four are local variables

D. `catherine` and `amos` are local variables, `_helen` and `_jeff` are instance variables

Diagnostic Question 2

```
public void danceDanceDance(CS15TA ta){
    ta.dance();
    ta.dance();
    ta.dance();
}
```

How do we get `CS15TA` Helen to dance dance dance?

A.
```
CS15TA helen = new CS15TA("Helen");  //CS15TA() takes a String argument
this.danceDanceDance(CS15TA helen);
```

B.
```
this.danceDanceDance(new CS15TA("Helen"));
```

C.
```
this.danceDanceDance(CS15TA helen = new CS15TA("Helen"));
```

D.
```
helen.danceDanceDance();
```

Diagnostic Question 3

Two versions of `fishMaker(…)` in class `Aquarium`, one of which is defined incorrectly:

```
//elided code
private Fish _fish;
public void fishMakerOne(Fish fish){
    fish = _fish;
}
public void fishMakerTwo(Fish fish){
    _fish = fish;
}
```

Which sets `_fish` to a `Clownfish`?

A. `this.fishMakerOne(new Clownfish());`

B. `this.fishMakerTwo(_fish);`

C. `fishMakerOne() = new Clownfish();`

D. `this.fishMakerTwo(new Clownfish());`

Diagnostic Question 4

The following are bodies of a method within the `SummerAdventure` class. We elided all the methods referred to in the bodies. Which of these displays bad style/inefficiency?

A.

```
Phineas phineas = new Phineas();
this.buildRocket(phineas);
Perry perry = new Perry();
this.doNothing(perry);
Candace candace = new Candace();
this.bustBrothers(candace);
```

B.

```
this.buildRocket(new Phineas());
this.doNothing(new Perry());
this.bustBrothers(new Candace());
```

C.

```
Doofenshmirtz doofenshmirtz = new Doofenshmirtz();
this.createEvilPlan(doofenshmirtz);
this.kidnapPerry(doofenshmirtz);
this.admitDefeat(doofenshmirtz);
```

D.

```
Phineas phineas = new Phineas();
Ferb ferb = new Ferb();
Buford buford = new Buford();
Isabella isabella = new Isabella();
phineas.singVocals();
ferb.playGuitar();
buford.playDrums();
isabella.harmonize();
```

Diagnostic Question 5

Within a method, it makes sense to store an instance of a class in a local variable when….

A. The variable is used only once
B. The variable is used throughout an entire class, in multiple methods
C. The variable is used in only that one method but multiple times within that method
D. The variable only needs to be created in the constructor and that's it

Explanation for 4 and 5

It is good practice to store a reference in a local variable when:

1. It is only used in a single method and therefore shouldn't be an instance variable
2. Additionally, the same reference is used multiple times as an argument

```
public void makeTodayAwesome(){
    Phineas phineas = new Phineas();
    this.brainstorm(phineas);
    this.makeBlueprints(phineas);
    this.buildProject(phineas);
}
```

3. Or a method needs to be called on the instance so you can't new it in the call because you don't have the reference to it

```
public void transformPerry(){
    Perry perry = new Perry();
    perry.wearFedora();
}
```

Diagnostic Question 6

Which `makeACake` method will execute all of its code correctly and succeed in baking a `Cake`?

A.

```
public Cake makeACake(){
    Cake cake = new Cake();
    return cake;
    cake.mix();
    cake.bakeAt350();
    cake.cool();
}
```

B.

```
public Cake makeACake(){
    Cake cake = new Cake();
    cake.mix();
    cake.bakeAt350();
    cake.cool();
    return cake;
}
```

C.

```
public Cake makeACake(){
    return cake;
    Cake cake = new Cake();
    cake.bakeAt350();
    cake.mix();
    cake.cool();
    return cake();
}
```

D.

```
public Cake makeACake(){
    if(1 == 1){
        return new Cake();
    } else {
        Cake cake = new Cake();
        cake.mix();
        cake.bakeAt350();
        cake.cool();
        return cake();
    }
}
```

Diagnostic Question 7

```
public class Scrabble {
    private int _score;

    public Scrabble(){
        _score = 0;
    }

    public int getScore(){
        return _score;
    }
}

public class ScrabblePlayer {
    public ScrabblePlayer(){
        this.keepScore();
    }
    public void keepScore(){
        Scrabble scrabble = new Scrabble();
        scrabble.getScore();
    }
}
```

Does `ScrabblePlayer` know what the score is?

A. Yes, because we called `scrabble.getScore()`
B. No, because the score is a local variable
C. Yes, because `Scrabble` is associated with `ScrabblePlayer`
D. No, `ScrabblePlayer` never stored the value of `getScore()` so the value was lost

Explanation of 7

- Because the code from the previous slide does not alter the value of `_score`, it is not particularly useful
- Illustrates an important point: calling an accessor method does not do anything unless we store for later use or immediately use that value somehow
- Examples of how we might use this:

Storing in a variable:

```
public void keepScore(){
    Scrabble scrabble = new Scrabble();
    int score = scrabble.getScore();
    score += 1;
    System.out.println(score);
}
```

Nesting in another call:

```
public void keepScore(){
    Scrabble scrabble = new Scrabble();
    //method from elsewhere in class
    this.calculateWinner(scrabble.getScore());
}
```

Diagnostic Question 8

```
public class CS15 {
    private HeadTA _headTA;
    // constructor ellided
    // other code elided
    public HeadTA getHTA(){
        return _headTA;
    }
}

public class HeadTA {
    //constructor, other code elided
    public void eatARawOnion() {
        //method body elided
    }
}

public class OnionEater {

    public OnionEater(){
        ???
    }
}
```

The TAs have defined a `getHTA()` method for you in the `CS15` class, which returns an instance of a `HeadTA`. They have also defined `eatARawOnion()` for you in the `HeadTA` class.

What is the best way to call `eatARawOnion()` in the `OnionEater` constructor?

A.
```
CS15 cs15 = new CS15();
HeadTA headTA = cs15.getHTA();
headTA.eatARawOnion();
```

B.
```
return new cs15.getHTA().eatARawOnion();
```

C.
```
CS15 cs15 = new CS15();
cs15.getHTA().eatARawOnion();
```

D.
```
newCS15().getHTA().eatARawOnion();
```

Explanation

- Good example of double dot notation/method chaining
- A is valid, but inefficient:

A:

```
CS15 cs15 = new CS15();
HeadTA headTA = cs15.getHTA();
headTA.eatARawOnion();
```

C:

```
CS15 cs15 = new CS15();
cs15.getHTA().eatARawOnion();
```

- We've saved a line of code!
  - And some memory space for a local variable which may not get re-used!
- Who would actually eat a raw onion, though?
- Oh…

Lecture 10

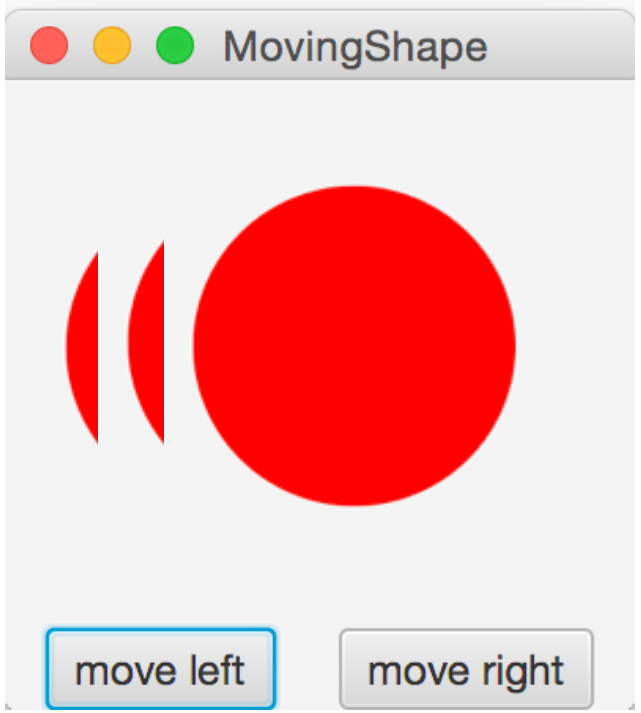
Graphics Part III – Building up to Cartoon

Outline

- Shapes
  - example: `MovingShape`
  - `App`, `PaneOrganizer`, and `MoveHandler` classes
- Constants
  - Clicker Question: Slide 44
- Composite Shapes
  - example: `Alien`
  - Clicker Question: Slide 56, Slide 64
- Cartoon

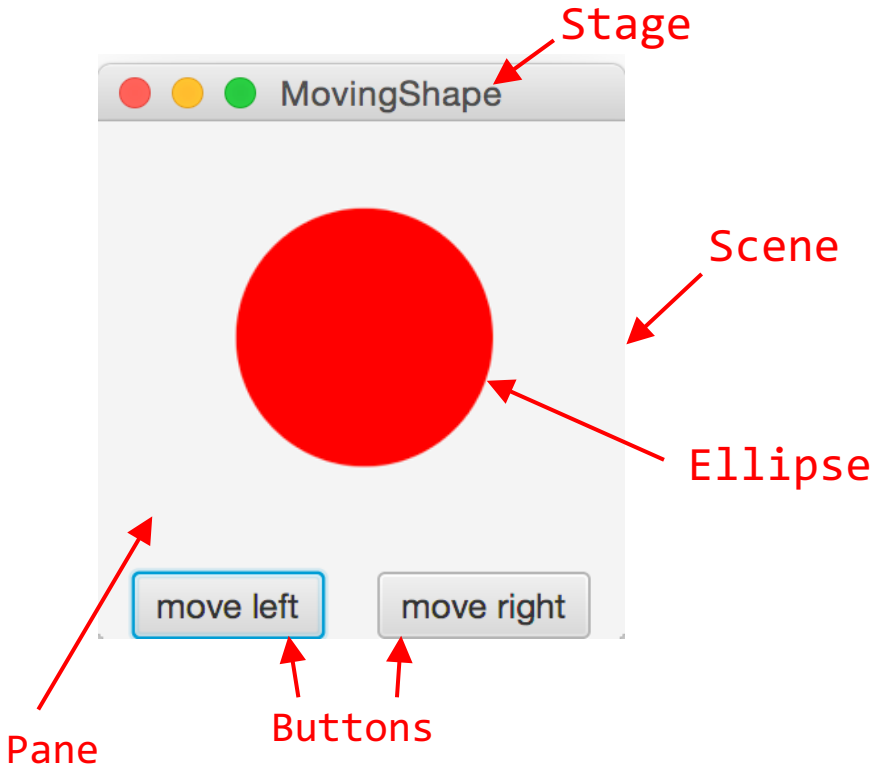
Example: MovingShape

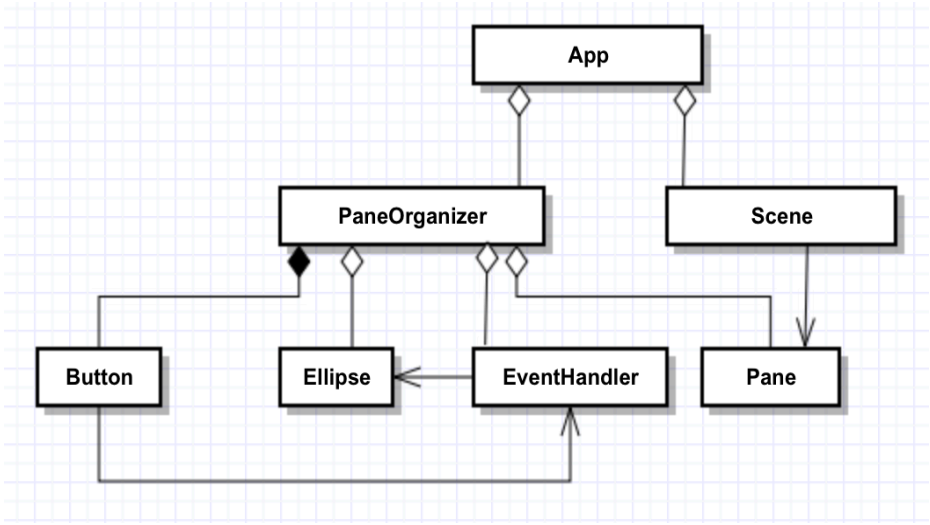
- Specification: App that displays a shape and buttons that shift position of the shape left and right by a fixed increment
- Purpose: Practice working with absolute positioning of Panes, various Shapes, and more event handling!



Process: MovingShapeApp

1. **Write 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





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`

```
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)

```
public class App extends Application {

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



    }

}
```

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`

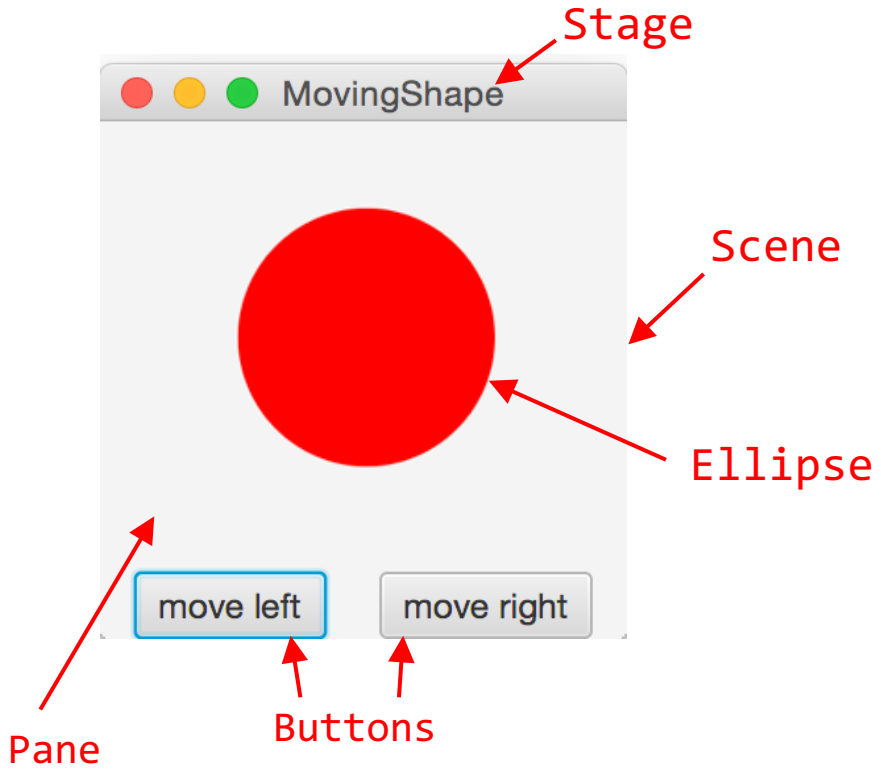
1b. Instantiate a `Scene`, passing in `organizer.getRoot()` and desired width and height of `Scene` (in this case 200x200)

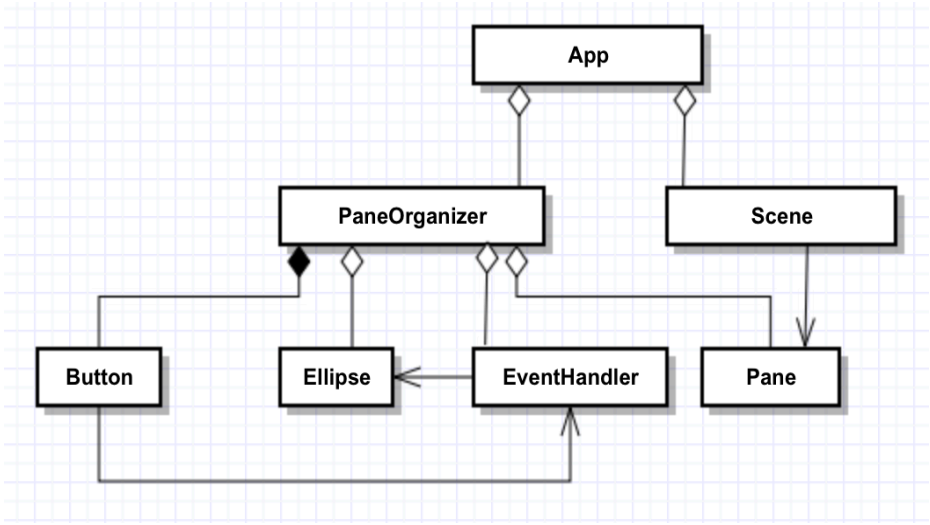
1c. Set `scene`, set `Stage`'s title and show it!

```
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 `Button`s with `EventHandler`s that handle `Button`s' `ActionEvent`s (clicks) by moving `Shape` correspondingly





MovingShapeApp: PaneOrganizer Class (1/4)

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 (2/4)

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

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

```
public class PaneOrganizer {
    private Pane _root;


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




    }

    public Pane getRoot() {
        return _root;
    }
}
```

MovingShapeApp: PaneOrganizer Class (3/4)

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

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

2c. Instantiate the `Ellipse` and add it as child of the root `Pane`

```
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;
    }
}
```

MovingShapeApp: PaneOrganizer Class (4/4)

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

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

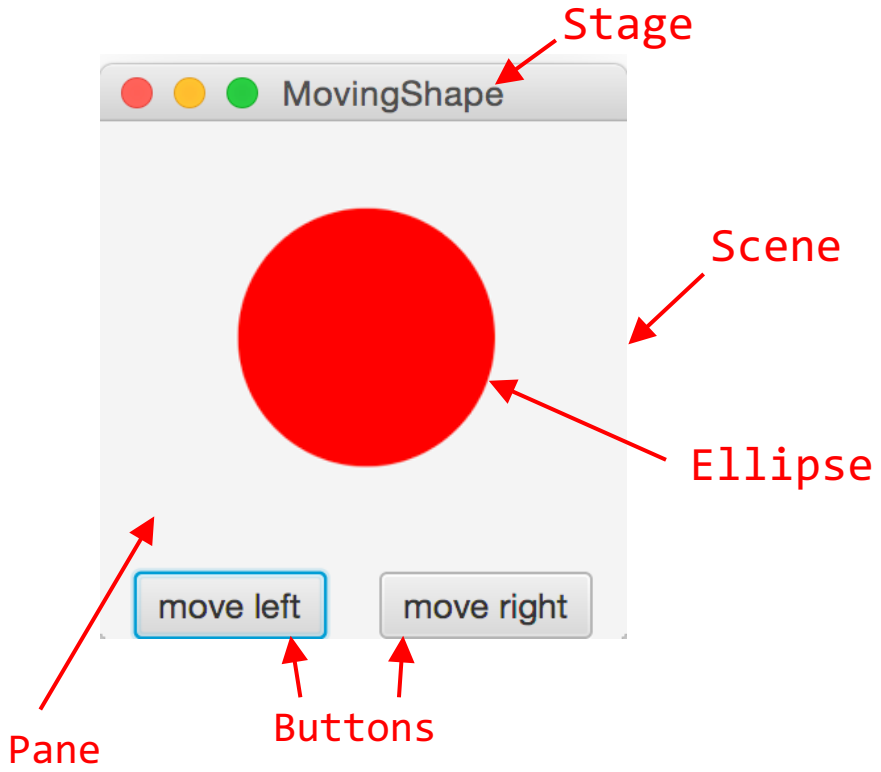
2c.Instantiate the `Ellipse` and add it as a child of the root `Pane`

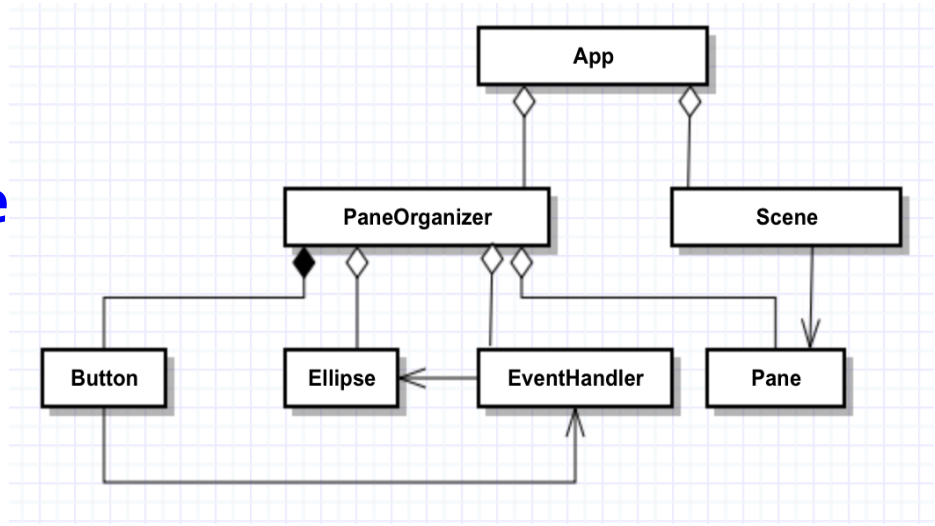
2d. Call `setupShape()` and `setupButtons()`, defined next

```
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 code for creating our custom `Pane`**
4. Register `Button`s with `EventHandler`s that handle `Button`s' `ActionEvent`s (clicks) by moving `Shape` correspondingly





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 `PaneOrganizer` constructor—it is good practice to **factor** out code into **helper methods** that are called within the constructor—another use of the delegation pattern
  - `setupShape()` fills and positions `Ellipse`
  - `setupButtons()` adds and positions `Button`s, and registers them with their appropriate `EventHandler`s
- Generally, helper methods should be `private` – more on this in a moment

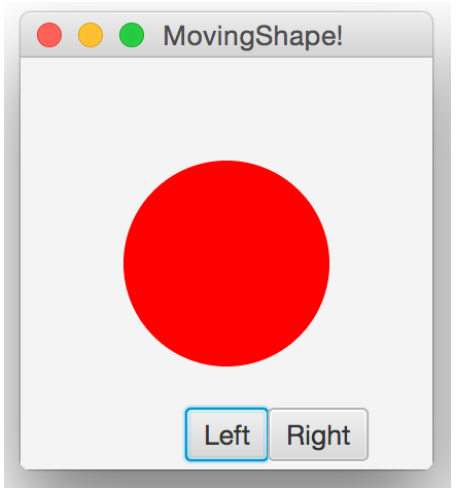
MovingShapeApp: setupShape() helper method

- 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?
  - 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 w/o the subclass knowing about them!

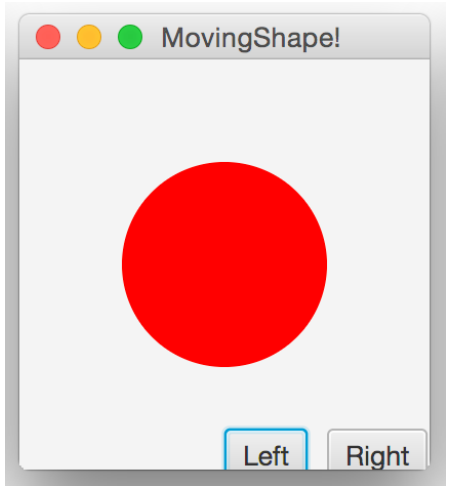
```
public class PaneOrganizer {
    private Pane _root;
    private Ellipse _ellipse;

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

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

    public Pane getRoot() {
        return _root;
    }

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

Aside: `PaneOrganizer` Class (1/3)

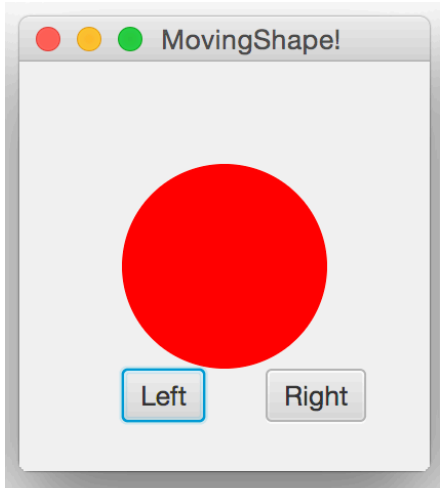
- We were able to absolutely position (position is fixed, cannot be changed) `_ellipse` in the root `Pane` because our root is simply a `Pane` and not one of the more specialized subclasses
- We could also use absolute positioning to position the `Button`s in the `Pane` in our `setUpButtons()` method… But look how annoying trial-and-error is!



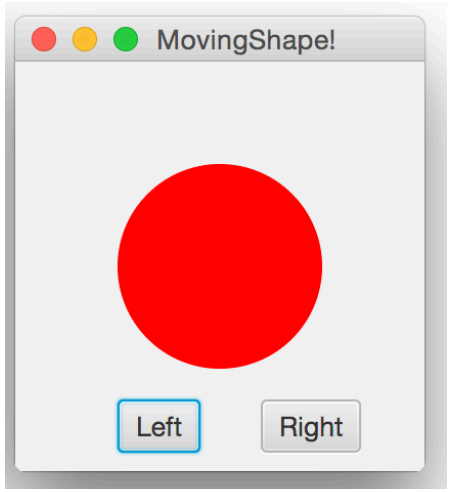

```
left.relocate(50,165);
right.relocate(120,165);
```




```
left.relocate(100,180);
right.relocate(150,180);
```




```
left.relocate(50,150);
right.relocate(120,150);
```

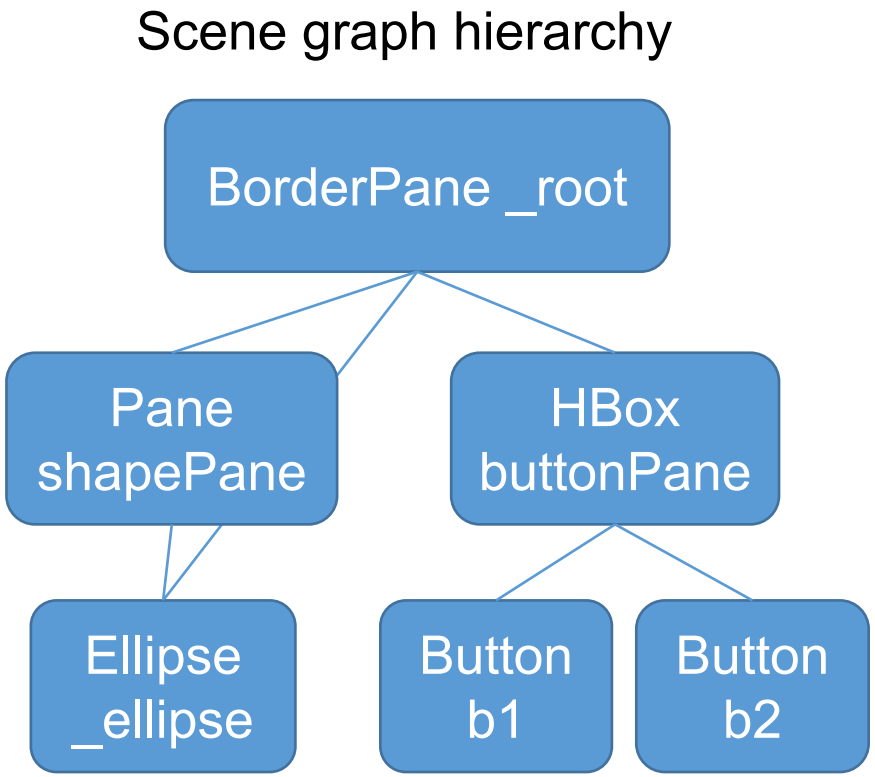



```
left.relocate(50,165);
right.relocate(120,165);
```

Is there a better way? …hint: leverage Scene Graph hierarchy and delegation!

Aside: `PaneOrganizer` Class (2/3)

- Rather than absolutely positioning `Button`s directly in root `Pane`, use a specialized layout `Pane`: add a new `HBox` as a child of the root `Pane`
  - add `Button`s 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
  - could simply add `Ellipse` to CENTER of root `BorderPane`
  - but this won't work—if `BorderPane` dictates placement of `Ellipse` we won't be able to update its position with `Button`s
  - instead: create a `Pane` to contain `Ellipse` and add the `Pane` as child of root!



Aside: `PaneOrganizer` Class (3/3)

- This makes use of the built-in layout capabilities available to us in JavaFX!
- Also makes symmetry between one panel holding a shape (in Cartoon you'll make composite shapes for such a panel) and the panel holding our buttons
- Note: this is only one of *many* design choices for this application!
  - keep in mind all of your different layout options when designing your programs!
  - 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

3a. Change root to a BorderPane, create a Pane to contain Ellipse

```
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!
    }

}
```

MovingShapeApp: update to BorderPane

3a. Change root to a BorderPane, create a Pane to contain Ellipse

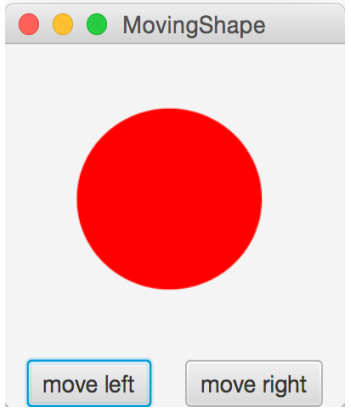
3b. To add shapePane to center of BorderPane, call setCenter(shapePane) on root

- note: none of the code in our setupShape() method needs to be updated 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** PaneOrganizer can still access it because it remains in its instance variable!
  - this could be useful if we want to change any properties of the Ellipse later on, e.g., updating it's x and y position, or changing its color
  - illustration of graphical vs. logical containment


```
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!
    }

}
```

MovingShapeApp: setupButtons() method (1/5)

3c. Make the buttons' Hbox: Instantiate a new HBox, then add it as child of BorderPane, in bottom position

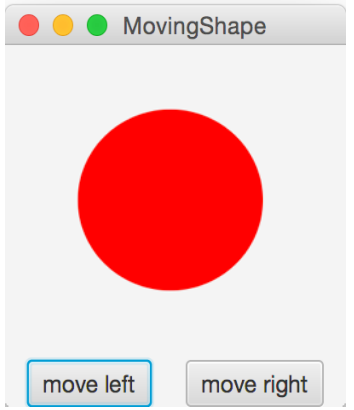



```
public class PaneOrganizer {
    private BorderPane _root;
    private Ellipse _ellipse;

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

        // setup of shape pane and shape elided!

        this.setupButtons();
    }

    private void setupButtons() {
        HBox buttonPane = new HBox();
        _root.setBottom(buttonPane);









    }

}
```

MovingShapeApp: setupButtons() method (2/5)

3c. Instantiate a new HBox, then add it as a child of BorderPane, in bottom position

3d. Instantiate two Buttons




```
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");




    }

}
```

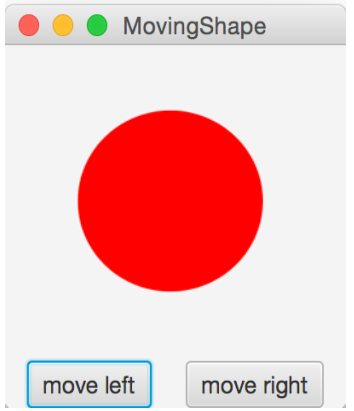
MovingShapeApp: setupButtons() method (3/5)

3c. Instantiate a new HBox, then add it as a child of BorderPane, in bottom position

3d. Instantiate two Buttons

3e. Add the Buttons as children of the new HBox

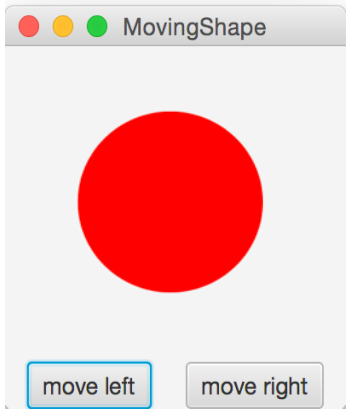
- note: different from before—now adding Buttons as children of HBox
- Some changes in the order are okay here, e.g., adding the buttons to the Hbox before adding it to the BorderPane
- Order **does** matter when adding children to Panes. For this HBox, b1 will be to the left of b2 because it is added first in the list of arguments in addAll(…)




```
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);



    }
}
```

MovingShapeApp: setupButtons() method (4/5)

3f. Set horizontal spacing between Buttons as you like

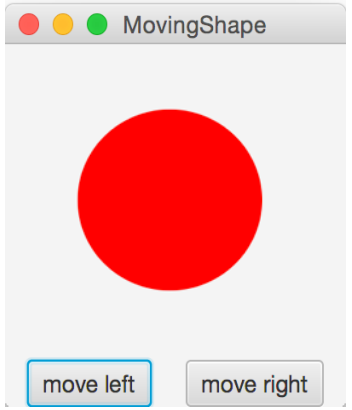



```
public class PaneOrganizer {
    private BorderPane _root;
    private Ellipse _ellipse;

    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);


    }

}
```

MovingShapeApp: setupButtons() method (5/5)

3f. Set horizontal spacing between Buttons as you like

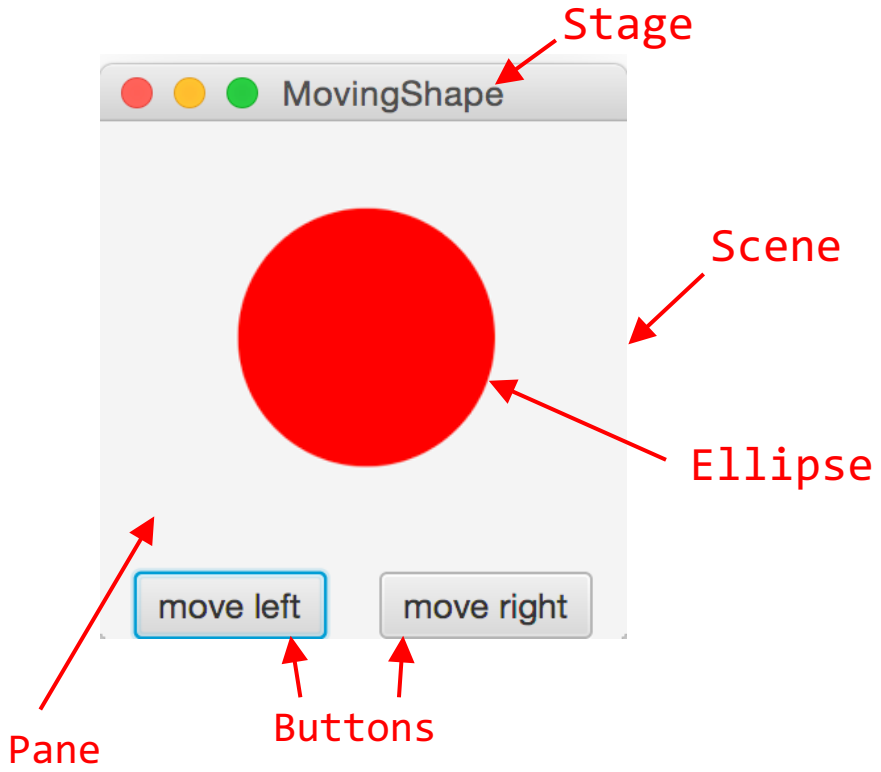
3g. Register Buttons with their EventHandlers by calling setOnAction() and passing in our instances of MoveHandler, which we will create next!

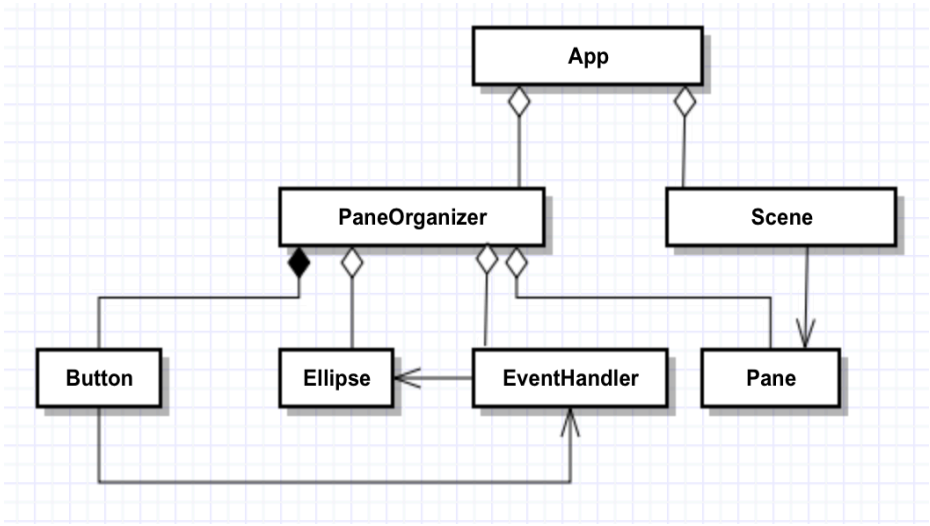



```
public class PaneOrganizer {
    private BorderPane _root;
    private Ellipse _ellipse;

    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);
        b1.setOnAction(new MoveHandler(true));
        b2.setOnAction(new MoveHandler(false));
    }

}
```

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 inner class `EventHandlers` that handle `Buttons'` `ActionEvents` (clicks) by moving `Shape` correspondingly**





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 `EventHandler`s, 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…

MovingShapeApp: MoveHandler (1/3)

4a. Declare an instance variable `_distance` that will be initialized differently depending on whether the `isLeft` argument is `true` or `false`

```
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) {

        }

    }

}
```

MovingShapeApp: MoveHandler (2/3)

4a. Declare an instance variable `_distance` that will be initialized differently depending on whether the `isLeft` argument is `true` or `false`

4b. Set `_distance` to 10 initially—if the registered `Button` `isLeft`, change `_distance` to -10 so the `Ellipse` moves in the opposite direction

```
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) {

        }

    }

}
```

MovingShapeApp: MoveHandler (3/3)

4a. Declare an instance variable `_distance` that will be initialized differently depending on whether the `isLeft` argument is `true` or `false`

4b. Set `_distance` to 10 initially – if the registered `Button` `isLeft`, change `_distance` to -10 so the `Ellipse` moves in the opposite direction

4c. Implement the handle method to move the `Ellipse` by `_distance` in the horizontal direction

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

    public PaneOrganizer() {
        // other code elided
    }

    private class MoveHandler implements EventHandler<ActionEvent> {
        private int _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);
        }

    }

}
```

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 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;
    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(left, right);
        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
} // end of PaneOrganizer 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

```
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;
}
```

Clicker 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.

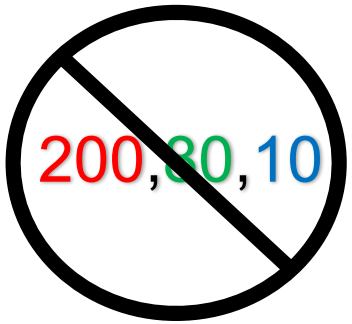
C. When the value is nontrivial (i.e., not 0 or 1)

D. All of the above.

The Whole App

no more literal numbers =
much better design!

Constants
class elided

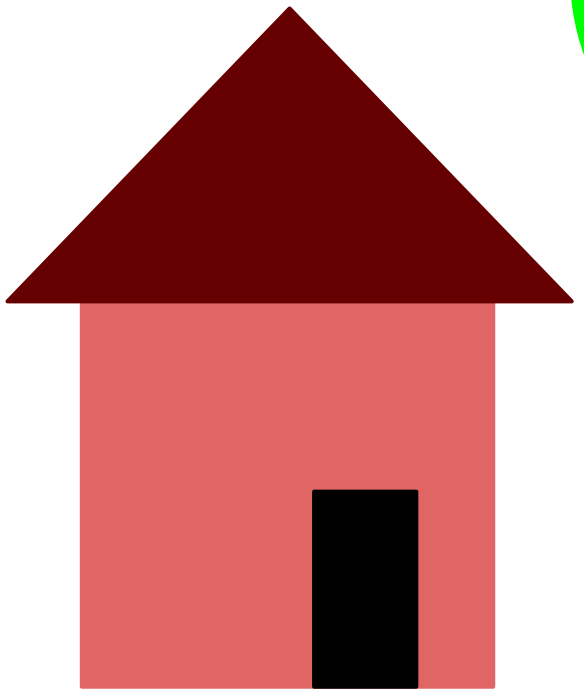


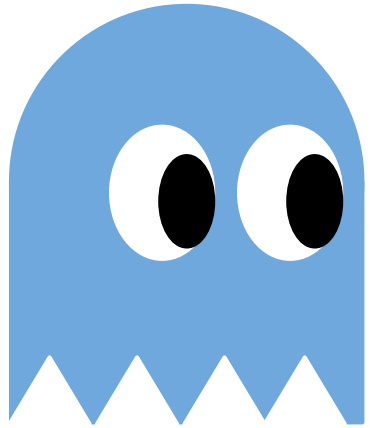

```
public class App extends Application {
    @Override
    public void start(Stage stage) {
        PaneOrganizer organizer = new PaneOrganizer();
        Scene scene = new Scene(organizer.getRoot(),
           Constants.APP_WIDTH, Constants.APP_HEIGHT);
        stage.setScene(scene);
        stage.setTitle("MovingShape!");
        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!



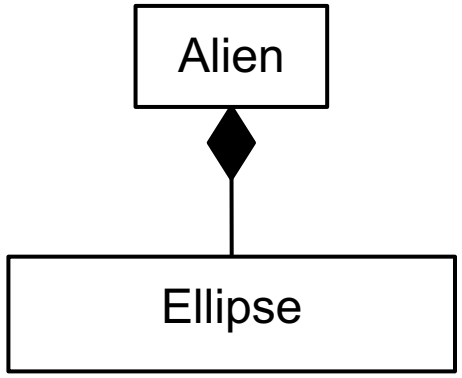


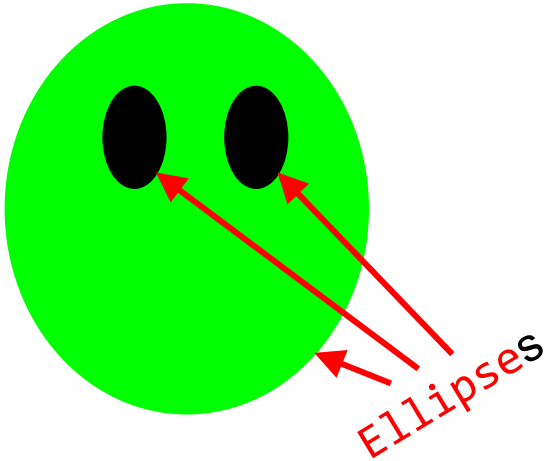
Spec: `MovingAlien`

- Transform `MovingShape` into `MovingAlien`
- An alien should be displayed on the central `Pane`, and should be moved back and forth by `Button`s

`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 `Ellipse`s)
  - 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 `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 `Ellipse`s—one for the face and one for each eye
- Constructor instantiates these `Ellipse`s, sets their initial sizes/colors, and adds them as children of the `alienPane`—which was passed in as a parameter
- The `Alien` class deals with each component of the composite shape individually
  - thus, must pass pane as a parameter to allow `Alien` class to define methods for manipulating composite shape in pane

```
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);

    }
}
```

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 (partially) on top of the earlier one. For this example, `_face` is added first, then `_leftEye` and `_rightEye` on top.

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 `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`:

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

- Because we called JavaFX's `getCenterX()` and `setCenterX(…)` on our shape from within the `PaneOrganizer` 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

```
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);
    }


}
```

MovingAlien: Alien Class (2/3)

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

```
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);
    }

    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!

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

    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();
    }
}
```

Clicker Question

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

A.

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

B.

```
public House () {
    //code to fill _foundation, _window, _door elided
    _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);
    new Pane().getChildren().addAll(_foundation, _window, _door);
    new Pane().setX(Constants.INITIAL_X_OFFSET);
}
```

C.

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

D.

```
public House (Pane housePane) {
    //code to fill _foundation, _window, _door elided
    _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);
    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 `PaneOrganizer`, such as *location setter/getter methods*!
3. **Modify `PaneOrganizer` to contain an `Alien` instead of an `Ellipse`**

MovingAlien: PaneOrganizer Class (1/4)

- 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!)

```
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(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(30);
        b1.setOnAction(new MoveHandler(true));
        b2.setOnAction(new MoveHandler(false));
    }

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

MovingAlien: PaneOrganizer Class (2/4)

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

```
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(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(30);
        b1.setOnAction(new MoveHandler(true));
        b2.setOnAction(new MoveHandler(false));
    }

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

MovingAlien: PaneOrganizer Class (3/4)

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

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

```
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(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(30);
        b1.setOnAction(new MoveHandler(true));
        b2.setOnAction(new MoveHandler(false));
    }

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

MovingAlien: PaneOrganizer Class (4/4)

- 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 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);
        }
    }
}
```

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("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 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);
        }
    }
}
```

Additional Classes

- Notice how we created another class for our Alien composite shape instead of simply adding each individual shape to `PaneOrganizer`
- As your programs get more complex (e.g., two shapes interacting with one another, shapes changing color, etc.), you may want to create even more additional classes that perform the desired functions instead of doing everything in `PaneOrganizer`
  - for example, if we are trying to create a Tic Tac Toe app, all of the game logic should go into a separate class; `PaneOrganizer` would only be responsible for placing `Pane`s and other elements on the screen
  - this will make `PaneOrganizer` less cluttered and your program as a whole much easier to read
  - keep this in mind for your upcoming assignments!!!

Clicker 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 Next Project: Cartoon! (1/2)

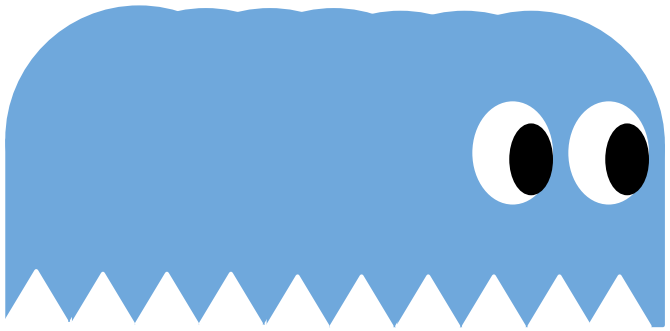
- You'll be building a JavaFX application that displays your own custom "cartoon", much like the examples in this lecture
- But your cartoon will be animated!

Your Next 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!
  - Early Handin: Tuesday, 10/16 at 11:59pm
  - On-Time Handin: Thursday, 10/18 at 11:59pm
  - Late Handin: Saturday, 10/20 at 11:59pm
- Section has 2 parts this week: Cartoon check-in and lab
  - Meet at normal section time at the Sunlab to get practice with JavaFX
  - Section TAs will send out signups for you to go over your design for Cartoon, and get to connect with your section TAs