Graphics Part II

Outline

- Layout Panes
- Absolute Positioning
- Shapes
  - example: MovingShape
- Constants
- Composite Shapes
  - example: MovingAlien

Layout Panes

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

HBox

- Similar to VBox—but lays everything out in a horizontal row (hence the name)
- Example:

  ```java
  // code for setting the scene elided
  HBox buttonBox = new HBox();
  Button b1 = new Button("Button One");
  Button b2 = new Button("Button Two");
  Button b3 = new Button("Button Three");
  buttonBox.getChildren().addAll(b1, b2, b3);
  ```
- Like VBox, we can set the amount of horizontal spacing between each child in the HBox using the `setSpacing(double)` method
**BorderPane (1/2)**

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

**Absolute Positioning**

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

**BorderPane (2/2)**

- Remember our VBox example from earlier?
- VBox `buttonBox = new VBox();`
  - Button `b1 = new Button("Top");` `b2 = new Button("Middle");` `b3 = new Button("Bottom");`
  - `buttonBox.getChildren().addAll(b1, b2, b3);`
  - `buttonBox.setSpacing(8);` `buttonBox.setAlignment(Pos.TOP_CENTER);`
- We can make our VBox the center of this **BorderPane**
  - `BorderPane container = new BorderPane();` `container.setCenter(buttonBox);`
- No need to use all of the regions—you could just use a few of them
- Unused regions are “compressed”, e.g. you could have a two-region (left/right) layout without a center

**Pixels and Coordinate System**

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

former HTA Sam Squires!
Pane

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

Creating Custom Graphics

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

**javafx.scene.shape Package**

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

**Shape Constructors**

- `Rectangle(double width, double height)`
- `Ellipse(double radiusX, double radiusY)`
- `Polygon(double ... points)`
  - The “...” signature means that you can pass in as many points as you would like to the constructor
  - Pass in points (even number of x and y coordinates) and `Polygon` will connect them for you
- Each of these `Shape` subclasses have multiple constructors, check out the JavaFX documentation for more options!
  - For example, if you wanted instantiate a `Rectangle` with a given position and size: `Rectangle(double x, double y, double width, double height)`

Example: `new Polygon(0, 10, 10, 10, 5, 0)`

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

- JavaFX Shapes have different behavior for setting their location within their parent’s coordinate system
  - Rectangle: use `setX(double)` and `setY(double)`
  - Ellipse: use `setCenterX(double)` and `setCenterY(double)`
  - Polygon: use `setLayoutX(double)` and `setLayoutY(double)`

- This is an inherited method; Polygon does not implement its own specific method for setting location unlike Rectangle and Ellipse.

JavaFX has many different ways to set location:
- From our experience, this is the most straightforward way.
- If you choose to use other methods, be sure you fully understand them or you may get strange bugs!
- Check out our JavaFX documentation and the Javadocs for more detailed explanations!

Shapes: Setting Size

- JavaFX Shapes also have different behavior for setting their size
  - Rectangle: use `setWidth(double)` and `setHeight(double)`
  - Ellipse: use `setRadiusX(double)` and `setRadiusY(double)`
  - Polygon: use `setScaleX(double)` and `setScaleY(double)`

  - Multiplies the original size in the X or Y dimension by the scale factor.
  - This is an inherited method; Polygon does not implement its own specific method for setting size.

- Again, this is not the only way to set size for Shapes, but it is relatively painless.
- Reminder: JavaFX documentation and Javadoc!

Accessors and Mutators of all Shapes

- Rotation:
  - Public final void setRotate(double rotate);
  - Public final double getRotate();

- Visibility:
  - Public final void setVisible(boolean visible);
  - Public final boolean getVisible();

- Color:
  - Public final void setStroke(Paint value);
  - Public final Paint getStroke();
  - Public final void setFill(Paint value);
  - Public final Paint getFill();

- Border:
  - Public final void setStrokeWidth(double val);
  - Public final double getStrokeWidth();

Example: MovingShape

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

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

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

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

4. Register `Buttons` with `EventHandlers` that handle `Buttons`' `ActionEvent`s (clicks) by moving `Shape` correspondingly

---

**1. Top-level Class: MovingShapeApp**

*NOTE: Exactly the same process as previous examples*

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

```java
public class App extends Application {
    public static void main(String[] args)
    
        Platform.startUp(new ApplicationLaunch[]{new MovingShapeApp()});
    }

    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();
    }

    public void setupShape() {
        // Code for creating custom Pane
    }

    public void setupButtons() {
        // Code for registering event listeners
    }
}
```

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

```java
public class App extends Application {
    public static void main(String[] args)
    
        Platform.startUp(new ApplicationLaunch[]{new MovingShapeApp()});
    }

    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();
    }
}
```

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

```java
public class App extends Application {
    public static void main(String[] args)
    
        Platform.startUp(new ApplicationLaunch[]{new MovingShapeApp()});
    }

    public void start(Stage stage) {
        PaneOrganizer organizer = new PaneOrganizer();
        Scene scene = new Scene(organizer.getRoot(), 200, 200);
        stage.setScene(scene);
        stage.setTitle("Moving Shape!");
        stage.show();
    }
}
```
Process: MovingShapeApp

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

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

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

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

(2) PaneOrganizer Class

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

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

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

```java
public class PaneOrganizer {
    private Pane _root;
    public PaneOrganizer() {
        _root = new Pane();
        _root.getChildren().add(new Ellipse(50, 50));
    }
    public Pane getRoot() {
        return _root;
    }
}
```
(2) PaneOrganizer Class

A. Instantiate the root Pane and store it in the instance variable _root
B. Create a public getRoot() method that returns _root
C. Instantiate the Ellipse and the Buttons, and add them as children of the root Pane
D. Call setupShape() and setupButtons(), defined next

```java
public class PaneOrganizer {
    private Pane _root;
    private Ellipse _ellipse;
    public PaneOrganizer() {
        _root = new Pane();
        _ellipse = new Ellipse(50, 50);
        _root.getChildren().add(_ellipse);
        this.setupShape();
        this.setupButtons();
    }
    public Pane getRoot() {
        return _root;
    }
}
```

(3) PaneOrganizer Class

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

(3) PaneOrganizer: setupShape() helper method

• For this application, “helper method” setupShape() will only set fill color and position Ellipse in Pane using absolute positioning
• Helper method is private—why is this good practice?
  o only the PaneOrganizer 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 methods may make use of them!
  o generally, helper methods should be private
**Digression: PaneOrganizer Class**

- We were able to absolutely position _ellipse_ in the root Pane because our root is simply a Pane and not one of the more specialized subclasses.
- We could also use absolute positioning to position the Buttons in the Pane.

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

**Digression: PaneOrganizer Class**

- Rather than absolutely positioning Buttons directly in root Pane, use a specialized layout Pane: add a new VBox as a child of the root Pane.
- Continuing to improve our design, use a BorderPane as root.
- Now need to add new Ellipse to the root.
  - Could simply add Ellipse to center of root but this won’t work—free let the BorderPane dictate the placement of the Ellipse we won’t be able to update the position of it using the Buttons.
  - Instead: create a Pane to contain Ellipse and add the Pane as child of root.

**Digression: PaneOrganizer Class**

- This makes use of the built-in layout capabilities available to us in JavaFX!
- Note: this is only one of many design choices for this application.
  - Keep in mind all of your different layout options when designing your programs!
  - Absolutely positioning the entire program is most likely not the best solution.

**(3) PaneOrganizer Class: update to BorderPane**

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

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

    public PaneOrganizer() {
        // 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 setupShape() {
        // more code to come!
    }

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

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

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

```java
public class PaneOrganizer {
    private BorderPane _root;
    private Ellipse _ellipse;
    public PaneOrganizer() {
        _root = new BorderPane();
        // setup shape pane
        Pane shapePane = new Pane();
        _ellipse = new Ellipse(50, 50);
        shapePane.getChildren().add(_ellipse);
        _root.setCenter(shapePane);
        this.setupShape();
        this.setupButtons();
    }
    private void setupButtons() {
        // more code to come!
    }
}
```

(3) PaneOrganizer Class: setupButtons() method

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

D. Instantiate two Buttons

```java
public class PaneOrganizer {
    private BorderPane _root;
    private Ellipse _ellipse;
    public PaneOrganizer() {
        _root = new BorderPane();
        // setup of shape pane and shape elided!
        this.setupShape();
    }
    private void setupButtons() {
        HBox buttonPane = new HBox();
        _root.setBottom(buttonPane);
        Button b1 = new Button("Move Left");
        Button b2 = new Button("Move Right");
        buttonPane.getChildren().addAll(b1, b2);
    }
}
```
(3) PaneOrganizer Class: setupButtons() method

F. Set horizontal spacing between Buttons as you like

```java
public class PaneOrganizer {
    private BorderPane _root;
    private Ellipse _ellipse;
    public PaneOrganizer() {
        _root = new BorderPane();
        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));
    }
}
```

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

Digression: Creating EventHandlers

• Our goal is to register each button with an EventHandler
  o the "Move Left" Button moves the Ellipse left by a set amount
  o 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...
  o why might this not be the optimal design?
  o remember, we want to be efficient with our code usage!

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

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

```java
A. public class MoveHandler {
    // other code elided
    public MoveHandler() {
        // other code elided
    }
}
```

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

```java
B. private class MoveHandler implements EventHandler<ActionEvent> {
    private int _distance;
    public MoveHandler(boolean isLeft) {
        // other code elided
    }
    public void handle(ActionEvent e) {
        // other code elided
    }
}
```

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

```java
C. private class MoveHandler implements EventHandler<ActionEvent> {
    private int _distance;
    public MoveHandler(boolean isLeft) {
        // other code elided
    }
    public void handle(ActionEvent e) {
        // other code elided
    }
}
```

---

**The Whole App**

```java
public class A {
    public class B {
        public class C {
            // other code elided
        }
    }
    public class D {
        // other code elided
    }
}
```
Creating Composite Shapes

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

Spec: MovingAlien

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

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

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

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

```java
_ellipse.moveCenterX(_ellipse.getXLoc() + _distance);
```

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

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

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

### Alien Class

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

   - Note: relative positions between the Ellipse's remains the same

B. Define `getXLoc()` method: the horizontal center of the Alien will always be center of face Ellipse
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 method!
3. Modify PaneOrganizer to contain an Alien instead of an Ellipse

PaneOrganizer Class

- setShape() method is no longer needed, as we now setup the Alien within the Alien class.
- only have to make a few changes to PaneOrganizer!
  - Instead of knowing about an ellipse called _ellipse, knows about an Alien called _alien
  - Change the shapePane to be an alienPane (we could have called it anything!)

PaneOrganizer Class

- setShape() method is no longer needed, as we now setup the Alien within the Alien class
  - remember that we set a default location for the Alien in its constructor.
PaneOrganizer Class

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

• We implemented setXLoc(...) and getXLoc() methods in Alien so MoveHandler can call them

Your Next Project: Cartoon!

• You’ll be building a JavaFX application that displays your own custom “cartoon”, much like the examples in this lecture

• But your cartoon will be animated!
Animation in Cartoon

- Use a **Timeline** to create incremental change
- It'll be up to you to figure out the details... but for each repetition of the **KeyFrame**, your cartoon should move a small amount!

--

Announcements

- Cartoon and Cartoon DQs have been released
  - Cartoon DQs are due this Friday 10/16 at 10pm
  - Read handouts carefully
    - **Read the handouts carefully**
- Review the Graphics lectures, the [Shapes Documentation](#), and the [JavaFX Guide](#)
- Lab 4 has been released