Graphics Part II
Outline

• Animation

• Layout Panes

• Absolute Positioning

• Shapes
Animation – Change Over Time

• Suppose we have an alien Shape we would like to animate (e.g. make him move across the screen)

• As in film and video animation, we 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 size, orientation, shape change, etc…

• How to orchestrate a sequence of incremental changes?
  o By coordinating them with a Timeline where change happens at defined instants
Introducing Timelines (1/2)

In our use of KeyFrame, the KeyFrame(s) work with the Timeline to control the animation (e.g. the Duration of the KeyFrame and the number of repetitions). The actual changing of variables to affect the animation is done in our EventHandler.
We don’t need full generality, and can do simple animation using a single KeyFrame that is repeated a fixed number of times or an indefinite number of times separated by a duration; each time EventHandler is called, it makes incremental changes to time-varying variables (e.g., \((x, y)\) position of a shape)
Using JavaFX Timelines (1/2)

- `javafx.animation.Timeline` is used to sequence one or more `javafx.animation.KeyFrames`, and optionally to run through them cyclically
  - Each `KeyFrame` lasts for its entire duration without making any changes, until its time interval ends and `EventHandler` is called to make updates

- When we instantiate a `KeyFrame`, we pass it
  - A `Duration` (e.g. `Duration.seconds(0.3)` or `Duration.millis(300)`), which defines time that each `KeyFrame` lasts
  - An `EventHandler` that defines what should occur upon completion of each `KeyFrame`

- `KeyFrame` and `Timeline` work together to control the animation, but our application’s `EventHandler` is the method that actually causes variables to change
Using JavaFX Timelines (2/2)

- We then pass our new KeyFrame into Timeline
- After we instantiate our Timeline, we must set its CycleCount property
  - this defines number of cycles in Animation
  - we will set cycle count to Animation.INDEFINITE, which will let Timeline run forever or until we explicitly stop it
- In order for Timeline to work, we must then call Timeline.play();
Another JavaFX App: **Clock**

- Simple example of a discrete (non-smooth) animation
- Specifications: App should display current date and time, updating every second

**Useful classes:**
- `java.util.Date`
- `javafx.util.Duration`
- `javafx.animation.KeyFrame`
- `javafx.animation.Timeline`
Process: Clock

1. **Write** App class that extends `javafx.application.Application` and implements start

2. Write a `PaneOrganizer` class that instantiates root node and returns it in a public `getRoot()` method. Instantiate a `Label` and add it as root node’s child. Factor out code for `Timeline` into its own method.

3. In our own `setupTimeline()`, instantiate a `KeyFrame` passing in a `Duration` and an instance of `TimeHandler` (defined later). Then instantiate `Timeline`, passing in our `KeyFrame`, and play `Timeline`.

4. Write private inner `TimeHandler` class that implements `EventHandler`-- it should know about a `Label` and update its text on every `ActionEvent`
Clock: App class

**Note: Exactly the same process as in ColorTextApp**

1. 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();
    }
}
```
Clock: App class

**Note: Exactly the same process as in ColorTextApp**

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

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

```java
public class App extends Application {
    @Override
    public void start(Stage stage) {
        PaneOrganizer organizer = new PaneOrganizer();
        Scene scene = new Scene(organizer.getRoot(), 200, 200);
    }
}
```
Clock: App class

**Note: Exactly the same process as in ColorTextApp**

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

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

3. Set the Scene, set the Stage’s title, and show the Stage!

```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();
    }
}
```
Process: Clock

1. Write App class that extends javafx.application.Application and implements start

2. Write a PaneOrganizer class that instantiates root node and returns it in a public getRoot() method. Instantiate a Label and add it as root node’s child. Factor out code for Timeline into its own method

3. In our own setupTimeline(), instantiate a KeyFrame passing in a Duration and an instance of TimeHandler (defined later). Then instantiate a Timeline, passing in our KeyFrame, and play the Timeline

4. Write a private inner TimeHandler class that implements EventHandler -- it should know about a Label and update its text on every ActionEvent
Clock: Our PaneOrganizer Class

A. In the PaneOrganizer class’ constructor, instantiate a root VBox and set it as the return value of a public getRoot() method

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

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

    public VBox getRoot() {
        return _root;
    }
}
```
Clock: Our PaneOrganizer Class

A. In the PaneOrganizer class’ constructor, instantiate a root VBox and set it as the return value of a public `getRoot()` method

B. Instantiate a Label and add it to the list of the root node’s children

```java
public class PaneOrganizer{
    private VBox _root;
    private Label _label;

    public PaneOrganizer(){
        _root = new VBox();
        _label = new Label();
        _root.getChildren().add(_label);
    }

    public VBox getRoot() {
        return _root;
    }
}
```
Clock: Our PaneOrganizer Class

A. In the PaneOrganizer class’ constructor, instantiate a root VBox and set it as the return value of a public getRoot() method

B. Instantiate a Label and add it to the list of the root node’s children

C. Call setupTimeline(); will define this method next!!

```java
public class PaneOrganizer{
    private VBox _root;
    private Label _label;

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

    public VBox getRoot() {
        return _root;
    }
}
```
Process: Clock

1. Write an App class that extends javafx.application.Application and implements start

2. Write a PaneOrganizer class that instantiates the root node and returns it in a publicgetRoot() method. Instantiate a Label and add it as the root node's child. Factor out code for Timeline into its own method.

3. In setupTimeline(), instantiate a KeyFrame, passing in a Duration and an instance of TimeHandler (defined later). Then instantiate a Timeline, passing in our KeyFrame, and play the Timeline.

4. Write a private inner TimeHandler class that implements EventHandler -- it should know about a Label and update its text on every ActionEvent
Clock: PaneOrganizer class- setupTimeline()

Within setupTimeline():

A. Instantiate a KeyFrame

- Want to update text of _label each second - therefore make Duration of the KeyFrame 1 second

Note: the KeyFrame calls TimeHandler’s handle method, which changes the label text before the next 1 second cycle starts.
**Clock:** PaneOrganizer class - setupTimeline()

**Within setupTimeline():**

A. Instantiate a KeyFrame

- Want to update text of _label each second - therefore make Duration of the KeyFrame 1 second
- For the EventHandler parameter pass an instance of our TimeHandler class, to be created later

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

    public void setupTimeline(){
        KeyFrame kf = new KeyFrame(
            Duration.seconds(1),
            new TimeHandler());
    }
}
```
public class PaneOrganizer{
    //other code elided
    public void setupTimeline(){
        KeyFrame kf = new KeyFrame(
            Duration.seconds(1),
            new TimeHandler());
        Timeline timeline = new Timeline(kf);
    }
}
Clock: PaneOrganizer class - setupTimeline()

Within setupTimeline():

A. Instantiate a KeyFrame

B. Instantiate a Timeline, passing in our new KeyFrame

C. Set the CycleCount to INDEFINITE

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

    public void setupTimeline(){
        KeyFrame kf = new KeyFrame(
            Duration.seconds(1),
            new TimeHandler());

        Timeline timeline = new Timeline(kf);

        timeline.setCycleCount(
            Animation.INDEFINITE);
    }

    }
```
Clock: PaneOrganizer class- setupTimeline()

Within setupTimeline():
A. Instantiate a KeyFrame
B. Instantiate a Timeline, passing in our new KeyFrame
C. Set CycleCount to INDEFINITE
D. Play, i.e. start Timeline

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

    public void setupTimeline(){
        KeyFrame kf = new KeyFrame(
            Duration.seconds(1),
            new TimeHandler());

        Timeline timeline = new Timeline(kf);
        timeline.setCycleCount(Animation.INDEFINITE);
        timeline.play();
    }
}
```
Process: Clock

1. Write an App class that extends javafx.application.Application and implements start

2. Write a PaneOrganizer class that instantiates the root Node and returns it in public getRoot() method. Instantiate a Label and add it as root node’s child. Factor out code for Timeline into its own method.

3. In setupTimeline(), instantiate a KeyFrame passing in a Duration and an instance of TimeHandler (defined later). Then instantiate a Timeline, passing in our KeyFrame, and play the Timeline.

4. Write a private inner TimeHandler class that implements EventHandler – it should know about a Label and update its text on every ActionEvent
Clock: TimeHandler Private Inner Class

A. The last step is to create our TimeHandler and implement handle(), specifying what to occur at the end of each KeyFrame – called automatically by j-fx

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

    private class TimeHandler implements EventHandler<ActionEvent>{
        public void handle(ActionEvent event){
            Date now = new Date();
            _label.setText(now.toString());
        }
    }

} //end of PaneOrganizer class
```
Clock: TimeHandler Private Inner Class

A. The last step is to create our `TimeHandler` and implement `handle()`, specifying what to occur **at the end** of each `KeyFrame` – called automatically by j-fx

B. `java.util.Date` represents a specific instant in time. `Date` is a representation of the time, to the nearest millisecond, at the moment the `Date` is instantiated

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

    private class TimeHandler implements EventHandler<ActionEvent>{
        public void handle(ActionEvent event){
            Date now = new Date();
        }
    }

    //end of private TimeHandler class

} //end of PaneOrganizer class
```
Clock: TimeHandler Private Inner Class

A. The last step is to create our TimeHandler and implement handle(), specifying what to occur at each KeyFrame — called automatically by j-fx

B. java.util.Date represents a specific instant in time. Date is a representation of the time, to the nearest millisecond, at the moment the Date is instantiated

C. Because our Timeline has a Duration of 1 second, each second a new Date will be generated, converted to a String, and set as the _label's text. This will appropriately update the Label with correct time every second!

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

    private class TimeHandler implements EventHandler<ActionEvent>{
        public void handle(ActionEvent event){
            Date now = new Date();
            _label.setText(now.toString());
        }
    }

    //end of private TimeHandler class
}
```

```java
} //end of PaneOrganizer class
```
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("Clock");
        stage.show();
    }
}

public class PaneOrganizer{
    private VBox _root;
    private Label _label;

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

    public VBox getRoot() {
        return _root;
    }

    public void setupTimeline(){
        KeyFrame kf = new KeyFrame(Duration.seconds(1),
                new TimeHandler());
        Timeline timeline = new Timeline(kf);
        timeline.setCycleCount(Animation.INDEFINITE);
        timeline.play();
    }

    private class TimeHandler
            implements EventHandler<ActionEvent>{
        public void handle(ActionEvent event){
            Date now = new Date();
            _label.setText(now.toString());
        }
    }
}
Layout Panes

• Until now, we have been adding all our GUI components to a **VBox**
  o **VBoxes** lay everything out in one vertical column

• What if we want to make some more interesting GUIs?

• Use different types of layout panes!
  o **VBox** is just one of many JavaFX panes—there are many more options
  o we will introduce a few, but check out our documentation or JavaDocs for a complete list
HBox

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

• Example:

  ```java
  // code for setting the scene elided
  HBox buttonBox = new HBox();
  Button b1 = new Button("Button One");
  Button b2 = new Button("Button Two");
  Button b3 = new Button("Button Three");
  buttonBox.getChildren().addAll(b1, b2, b3);
  ```

• Like VBox, we can set the amount of horizontal spacing between each child in the HBox using the setSpacing(double) method
BorderPane (1/2)

- **BorderPane** lays out children in top, left, bottom, right and center positions
- To add things visually, use `setLeft(Node), setCenter(Node),` etc.
  - this includes an implicit call to `getChildren().add(...)`
- Use any type of **Node**—**Panes** (with their own children), **Buttons**, **Labels**, etc.
BorderPane (2/2)

- Remember our VBox example from earlier?

```java
VBox buttonBox = new VBox();
Button b1 = new Button("Top");
Button b2 = new Button("Middle");
Button b3 = new Button("Bottom");
buttonBox.getChildren.addAll(b1,b2,b3);
buttonBox.setSpacing(8);
buttonBox.setAlignment(Pos.TOP_CENTER);
```

- We can make our VBox the center of this BorderPane

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

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

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

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

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

• Screen is a grid of **pixels** (tiny squares, each with RGB components)

• Cartesian plane with:
  - origin in upper-left corner
  - x-axis increasing left to right
  - y-axis increasing top to bottom
  - corresponds to English writing order
**Pane**

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

• We’ve now introduced you to using JavaFX’s native UI elements
  o ex: Label and Button

• Lots of handy widgets for making your own graphical applications!

• What if you want to create your own custom graphics?

• This lecture: build your own graphics using the javafx.scene.shape package!
JavaFX 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 “…” in the signature means that you can pass in as many points as you would like to the constructor
  - pass in Points (even number of x and y coordinates) and Polygon will connect them for you
  - passing points will define and position the shape of Polygon - this is not always the case with other Shapes (like Rectangle or Ellipse)
  - Example: new Polygon(0,10,10,10,5,0)

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

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

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

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

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

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

- **Visibility:**
  - `public final void setVisible(boolean visible);`
  - `public final boolean getVisible();`

- **Color:**
  - `public final void setStroke(Paint value);`
  - `public final Paint getStroke();`
  - `public final void setFill(Paint value);`
  - `public final Paint getFill();`

- **Border:**
  - `public final void setStrokeWidth(double val);`
  - `public final double getStokeWidth();`

*Final = can't override method*

Rotation is about the `center` of the **Shape**

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

Generally, use a **Color**, which inherits from **Paint**. Use predefined color constants `Color.WHITE`, `Color.BLUE`, `Color.AQUA`, etc., or define your own new color by using the following syntax:

```
Pset color = Color.color(0.5, 0.5, 0.5);
```

OR:

```
Pset color = Color.rgb(100, 150, 200);
```
Announcements

• TASafeHouse is due this weekend!
  o Early handin: this Friday (10/7) at **10 PM**
  o Ontime handin: this Sunday (10/9) at **11:59 PM**
  o There is no late deadline for this assignment!

• Cartoon out on Tuesday!
  o No design discussion - individual check-in with discussion TA instead
  o TAs will reach out by email by end of day on Monday

• Review Session today at 7:30pm-9pm, MacMillan 115

• Have a great long weekend!