Top Hat Question

Which of the following is the correct way to create a private inner class that uses the EventHandler interface?

A. `public class PaneOrganizer {
   private class MoveHandler extends EventHandler {
      public MoveHandler() {}  
      public void handle(ActionEvent event) {} 
   }  
}`

B. `public class PaneOrganizer {
   private class MoveHandler implements EventHandler {
      public MoveHandler() {}  
      public void handle(ActionEvent event) {} 
   }  
}`

C. `public class PaneOrganizer {
   private class MoveHandler implements EventHandler<ActionEvent> {
      public MoveHandler() {}  
      public void handle(ActionEvent event) {} 
   }  
}`

Lecture 9

Graphics Part II – Understanding Animations & Shapes

Outline

- Animation
- Layout Panes
- Java FX Shapes
Animation – Change Over Time

- Suppose we have an alien shape we would like to animate (e.g., make it 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?

Introducing Timelines (1/3)

- The Timeline sequences one or more KeyFrames.
  - A KeyFrame can be thought of as a singular photo.
  - Each KeyFrame lasts for its entire Duration without making any changes.
  - When the Duration ends, the EventHandler updates variables to affect the animation.

Introducing Timelines (2/3)
Introducing *Timelines* (3/3)

We can do simple animation using a single *KeyFrame* that is repeated a fixed or indefinite number of times. *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.KeyFrame`s or run through them cyclically
  - each *KeyFrame* lasts for its entire *Duration* until its time interval ends and *EventHandler* is called to make updates
- When we instantiate a *KeyFrame*, we pass in
  - 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 what 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
  - defines number of cycles in *Animation*
  - setting cycle count to `Animation.INDEFINITE` 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 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 (Stage)
2. Write a PaneOrganizer class that instantiates the 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 (1/3)
*Note: Exactly the same process as in ColorChanger’s App (Lecture 8)*
1a. Instantiate a PaneOrganizer and store it in the local variable organizer

```java
public class App extends Application {
    @Override
    public void start(Stage stage) {
        PaneOrganizer organizer = new PaneOrganizer();
        Organizer organizer = new PaneOrganizer();
    }
```
**Clock: App class (2/3)**

*Note: Exactly the same process as in ColorChanger’s App [Lecture 8]*

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`.

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

*Note: Exactly the same process as in ColorChanger’s App [Lecture 8]*

**Clock: App class (3/3)**

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 the `Scene`.

1c. Set the `Scene`, set the `Stage`'s title, and show the `Stage`.

```java
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();
    }
```

**Process: Clock**

1. Write an `App` class that extends `javafx.application.Application` and implements `start(Stage)`.

2. Write a `PaneOrganizer` class that instantiates the root node and returns it in a `getRoot()` method. Instantiate a `Label` and add it as root node's child. Factor out code for `Timeline` into its own method, which we'll call `setupTimeline()`.

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`.
2a. 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;
    private Label _label;
    public PaneOrganizer()
    { 
        _root = new VBox();
        _label = new Label();
        _root.getChildren().add(_label);
    }
    public VBox getRoot() {
        return _root;
    }
}
```

2b. Instantiate a Label and add it to the list of the root node’s children

```java
2a. In the PaneOrganizer class’ constructor, instantiate a root VBox and set it as the return value of a public getRoot() method

```

2c. Call setupTimeline(); this is another example of delegation to a specialized “helper method” which we’ll define next!

```java
2a. In the PaneOrganizer class’ constructor, instantiate a root VBox and set it as the return value of a public getRoot() method

```
Process: Clock

1. Write an App class that extends javafx.application.Application and implements start(Stage)
2. Write a PaneOrganizer class that instantiates the root node and returns it in a public getRoot() 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() (1/4)

Within setupTimeline():

3a. Instantiate a KeyFrame, which takes two parameters
   ○ want to update text of _label each second — therefore make Duration of the KeyFrame 1 second

public class PaneOrganizer{
    // other code elided
    public void setupTimeline(){
        KeyFrame kf = new KeyFrame(
            Duration.seconds(1), // how long
            new ActionHandler()
        )
    }
}
Within `setupTimeline()`:

3a. Instantiate a `KeyFrame`, which takes two parameters:
   - 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());
  }
}
```

Note: JavaFX automatically calls `TimeHandler`'s `handle()` method at end of `KeyFrame`, which in this case changes the label text, and the loop starts from the beginning of the `KeyFrame`.

Within `setupTimeline()`:

3a. Instantiate a `KeyFrame`
3b. Instantiate a `Timeline`, passing in our new `KeyFrame`
3c. Set `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);
    timeline.play();
  }
}
```
**Clock: PaneOrganizer class - setupTimeline() (4/4)**

Within setupTimeline():

3a. Instantiate a KeyFrame

```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();
    }
}
```

3b. Instantiate a Timeline, passing in our new KeyFrame

```java
Timeline timeline = new Timeline(kf);
```

3c. Set cycle count to INDEFINITE

```java
timeline.setCycleCount(Animation.INDEFINITE);
```

3d. Play, i.e. start Timeline

```java
timeline.play();
```

**Process: Clock**

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

2. Write a PaneOrganizer class that instantiates the root Node and returns it in public root() 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 (1/3)**

4a. The last step is to create our TimeHandler and implement handle(), specifying what should occur at the end of each KeyFrame — called automatically by JFX

```java
public class PaneOrganizer {
    // other code elided
    private class TimeHandler implements EventHandler<ActionEvent> {
        public void handle(ActionEvent event){
        }
    }
}
```
The last step is to create our `TimeHandler` and implement `handle()`, specifying what should occur at the end of each KeyFrame — called automatically by JFX.

4b. `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();
            // _label instantiated in constructor of PO
            _label.setText(now.toString());
        }
    }
    // end of private TimeHandler class
}
```

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 `_label` with correct time every second!
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 Java Docs for a complete list

HBox

• Similar to VBox, but lays everything out in a horizontal row (hence the name)
• Example:
  ```java
  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!
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

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!

Pane

• Pane allows you to lay things out completely freely, like on an art canvas – DIY graphics! More control, more work ;)
• It is a concrete superclass to all more specialized layout panes seen earlier that do automatic positioning
  o 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
  o 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
  - ex: Label and Button
- Lots of handy widgets for making your own graphical applications!
- What if you want to create your own custom graphics?
- This lecture: build your own graphics using the `javafx.scene.shape` package!

`javafx.scene.shape` Package

- JavaFX provides built-in classes to represent 2D shapes, such as rectangles, ellipses, polygons, etc.
- All these classes inherit from abstract class Shape, which inherits from Node:
  - 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
  - example: new Polygon(0,10,10,10,5,0)

- Each of these Shape subclasses have multiple overloaded constructors (see Math and Making Decisions, slide 58) — check out the JavaFX documentation for more options!
  - for example, if you wanted to instantiate a Rectangle with a given position and size: `Rectangle(double x, double y, double width, double height)`

Default position for Shape with this constructor would be (0,0)
**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, these are the most straightforward ways:
    - 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 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](#) and [Javadocs](#)!

**Accessors and Mutators of all Shapes**

- **Rotation:**
  - Use `getRotate()` and `setRotate(double)`.
  - Rotation is about the **center** of the shape's “bounding box”.
  - The **stroke** is the border that outlines the shape, while the **fill** is the color of the interior of the shape.

- **Visibility:**
  - Use `setVisible(boolean)` and `setVisible(boolean)`.

- **Color:**
  - Use `getStroke(boolean)` and `getStroke(boolean)`.
  - Use `setStroke(Paint value)` and `setFill(Paint value)`.

- **Border:**
  - Use `getStrokeWidth(double)` and `getStrokeWidth(double)`.

**Note:**

- Use predefined color constants `Color.WHITE`, `Color.BLUE`, `Color.AQUA`, etc., or define your own new color by using the following syntax:
  ```java
  Paint color = Color.color(0.5, 0.5, 0.5);
  ```
  OR:
  ```java
  Paint color = Color.rgb(100, 150, 200);
  ```
Announcements

- FruitNinja deadlines:
  - early: Friday, 10/4 at 11:59pm
  - on-time: Sunday, 10/6 at 11:59pm
  - late: Tuesday, 10/8 at 11:59pm
- Section Slides released after the end of the last section

HOSTED BY
DIVERSITY ADVOCATES

SPONSORED BY
UFLi CENTER

UFLi STEM Mixer

"Open to all! Even if you don’t plan to concentrate"

Monday, October 7th
2nd Floor Sciences Library
1-4:30pm