Lab 4: Introduction to JavaFX

This lab will give you a taste of building your own graphical applications using JavaFX. First you'll use **`Stages`**, **`Scenes`**, **`Panes`**, **`Rectangles`**, and more to create an application that visually matches our mock-up shown below. Next, you'll write **`EventHandlers`** that allow your application to respond to user input. Andy's Graphics lectures will be a useful reference throughout this lab, so be sure to take a look at them.

Goal: Create a JavaFX application that matches the below mock-up that lets users change the colors of the rectangles to new, random colors based on keyboard input. The quit button at the bottom should quit the application..

You should be familiar with partner coding by now, but remember that only one person should be typing at one time. Choose one partner to be the navigator and one to be the driver. Don't worry, you will swap roles at each checkpoint!

As always, begin this lab by perusing the reading with your partner!

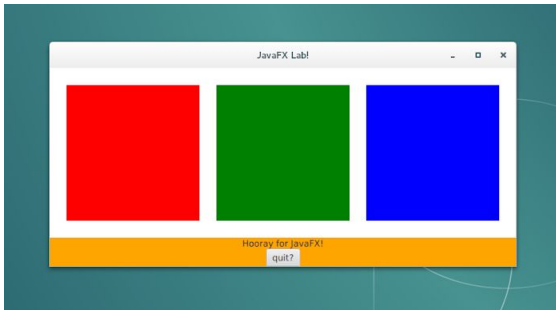
Part 0: Setting up Eclipse

Note: For this part of the lab *only*, you and your partner should both complete these steps on your own machines, although you should do so collaboratively. Once you both have Eclipse set up, complete the rest of the lab as a pair as we have done previously this year.

- Set up Eclipse by following the instructions linked to the reading.

Checkpoint 0: Call over a TA and show your working Eclipse! Remember that both partners must show a working Eclipse to move on. If both partners must share one computer due to space, the other partner may set up Eclipse at the end — but it is a prerequisite for being checked off for this lab!

Part 1: Building a GUI with JavaFX Panes



Getting Started

First, you'll be building the GUI pictured above from scratch!

- Run the script **`cs0150_install lab4`** to install the stencil code for this lab.
- Open up the **`lab4`** directory in Eclipse. You should see two stencil files: **`App.java`** and **`Constants.java`**. Open up **`App.java`**!
- The **`App`** class will be the top-level class for your whole program. Its job is to set up the outermost graphical container (a **`Stage`**).
- In the **`start(...)`** method, set a title for your **`Stage`** passed in as a parameter to your start method using the **`Stage`** class' **`setTitle(String s)`** method.
- Try compiling and running the program. You'll probably notice that nothing changes, and no **`Stage`** appears. It turns out that a **`Stage`** won't show up unless it's told that it should be shown. Call the **`show()`** method on your **`Stage`** to make sure it shows up.
- Run the program again. You should see a small **`Stage`** pop up in the top-left corner of your screen with a grey background. You can resize the **`Stage`** by clicking and dragging its bottom-right corner.

Adding Panes

Next, we want to add some content to our Stage. As you can see above, we want to create two sections, or **`Pane`**s: one with rectangles, and one with a label and quit button. Theoretically, we can add all of this directly to the **`Stage`**, but that could be an organizational nightmare with more complicated projects. Instead, we'll create a **`PaneOrganizer`** class, which will keep track of the 2 **`Pane`**s that we need

- Create a new class within the **`lab4`** package. Name this class "**`PaneOrganizer`**". Don't forget that even though the name of the *class* is **`PaneOrganizer`**, the name of the *file* should be **`PaneOrganizer.java`**

In our **`PaneOrganizer`**, we'll be creating a couple of **`javafx.scene.layout.Pane`**s and filling their contents.

- First, we want an object that's capable of laying out JavaFX scene objects in a nice, organized way. A **`BorderPane`** is a JavaFX organizational class that allows you to lay out certain objects on the left, right, top, bottom, and center of your application. Create an instance variable of type **`BorderPane`** in your **`PaneOrganizer`** class and instantiate it in the **`PaneOrganizer`** constructor. Hover over "BorderPane" to easily import the BorderPane class.
- Now write a method with the signature "**`public Pane getRoot()`**" that returns the **`BorderPane`** you created.

Now we can add the **`BorderPane`** to our **`Stage`**. To do this, we will create a **`Scene`**. You can think of a **`Scene`** as being a container for all GUI items. In CS15, you'll only need one **`Scene`** per application.

- Go back to the file **`App.java`**. In the **`start()`** method, instantiate a **`PaneOrganizer`** and add the **`BorderPane`** you've just created to a new **`Scene`** by calling:

```
PaneOrganizer organizer = new PaneOrganizer();

Scene scene = new Scene(organizer.getRoot());

primaryStage.setScene(scene);
```

- **Note:** Make sure you do all of this before the line where you show the **`Stage`**!

Are you there, Pane?

If you run the program now and expand the **`Stage`**, it looks like nothing has changed. How do we know that our **`BorderPane`** is even there? Let's make sure everything's working by giving our **`BorderPane`** a background color by calling the method **`setStyle()`** on it.

`Panes` rely on CSS for much of their styling. Colors in CSS can be written as a predefined color name or a hex code. A hex code is a "#" character followed by six hexadecimal digits (for more information, see this page). Therefore, setting the background color to **`orange`** (**`#FFA500`**) can be written one of two ways:

1. **`_pane.setStyle("-fx-background-color: orange;");`**
2. **`_pane.setStyle("-fx-background-color: #FFA500;");`**

- Set the background color of your **`BorderPane`** to orange in the constructor of **`PaneOrganizer.`**
- Now, when you run the program and expand the **`Stage`** by clicking and dragging, the window should be filled in orange. That means our **`BorderPane`** is displaying and everything is working properly so far.
- If you're not seeing orange, you've got some debugging to do!
    - **TIP:** If you get an error that reads "**`unmappable character for encoding ASCII`**", try rewriting the code rather than copying it from the PDF.

Create and Size Sub-Panes

Now that we have a **`PaneOrganizer`** and a **`BorderPane`**, we're well on our way to adding the rectangles. The **`BorderPane`** is our overall organization, but we still need to fill it in with the 2 **`Pane`**s we discussed before: one is the area with the rectangles, and the other contains the label for our app. In order to keep our GUI and program completely organized, we need to create two

sub-Panes to represent these two areas. Let's make the top **`Pane`**, which contains the rectangles, first.

- Talk to your partner about the difference between public and private methods, and come up with one more private method that Perry could have in the **`stopDoof()`** method. Write your answer in a new text file and be prepared to show it to a TA for checkoff.

To add rectangles to our app, we will be making use of private methods and classes, which we reviewed in the reading for this lab.

Write a new private method in your **`PaneOrganizer`** called **`createRectsPane()`** that creates an instance of the **`Pane`** class and adds it to your **`BorderPane`**. This method should not return anything. To do this, follow these instructions:

- At the beginning of this method, create a new **`Pane`** (call it **`rectsPane`**) and set its size using the **`setPrefSize()`** method, passing in the dimensions given in the **`Constants`** class for with width and height.
- Set the background color for the **`rectsPane`** in the same way you did for the **`BorderPane`**, but this time color it white (**`#FFFFFF`**).
- Add the **`Pane`** you just made to your **`BorderPane`** using the **`setTop(...)`** method.

  Note: If you are doing this lab over ssh, the initial size of the stage may be incorrect--drag the bottom-right corner to expand the window.
- In the constructor for your **`PaneOrganizer`**, after instantiating your **`BorderPane`**, call your **`createRectsPane()`** method.
- Run your program. You should now see the top **`Pane`** show up!

Checkpoint 1: Compare your **`PaneOrganizer`** class with your neighbors. Discuss any differences you find and why those differences exist.

- When you have finished this, switch roles with your partner: the navigator should now be the driver, and vice versa.

Adding Rectangles

Now that we know that our top **`Pane`** has been added, let's add some **`Rectangle`**s to it. We want special **`Rectangle`**s that will support keyboard entry to change color. Let's make a new class called **`KeyableRect`**, which will contain a **`Rectangle`** shape and give it some special capabilities.

Overarching Design:

The only thing we can add to a **Pane** is a **Node**, so in order to add this **KeyableRect** to the pane, we need to create a getter, **getNode()**, which would return **KeyableRect**'s **Rectangle**. In **PaneOrganizer,** you would call **getNode()** on the **KeyableRect** instance and add or remove that directly from the **Pane**'s children list.

What does this design sound like in plain English? *"When the **PaneOrganizer** wants to add a **KeyableRect** to its **Pane**, it will get the **Node** from **KeyableRect** and add it to its **Pane**".*

Keep in Mind: there are other ways to add rectangles to the pane, but this design is the most straightforward and will simply code a lot when our projects get more complicated (keep this in mind for **DoodleJump**!)

Now let's code this design out:

- Create a class called **KeyableRect**, which will contain an instance variable of type **Rectangle**.
- In its constructor, instantiate a **Rectangle** for that instance variable.
- Write a getter method to return the rectangle **Node** from this **KeyableRect**. Hint: this method should probably return an object of type **Node**. Remember that practically all JavaFX objects are **Node**s. Polymorphism!!

Because all **KeyableRect**s will have the same size but different locations and colors, we can specify the location and color as arguments to the **KeyableRect** constructor!

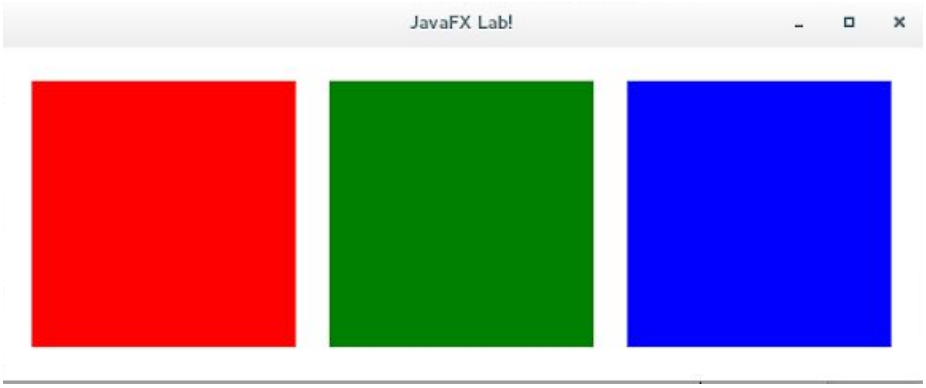
- Modify the **KeyableRect**s constructor to take in a **double** for the x location, a **double** for the y location, and a **Color**.

  Note: this color is a JavaFX class **Color**, not a CSS color as described above.

- Then, set the size of your **Rectangle** using the width and height constants provided in the **Constants** class, as well as its location and color using the values from the **KeyableRect**'s constructor. Refer to the Graphics II lecture or JavaFX Shapes Documentation for more information about how to accomplish this.
- Go back to the **createRectsPane()** method.
  - Declare and instantiate 3 **KeyableRects** called **leftRect**, **centerRect**, and **rightRect**, setting the location of each (using the values we've provided in the **Constants** class) and the color via the constructor.
- Run your program - where are your **KeyableRect**s? Much like your **RectsPane** not showing up until it was added as a child of the **BorderPane**, we need to add the **KeyableRect**s to the **Pane** as elements of the scene graph.
- To add the **Rectangle**s as children of the **Pane**, we need to get the list of children belonging to the **Pane** you made earlier in **createRectsPane** and add them all to it.
  - After instantiating your Rectangles, call **rectsPane.getChildren().addAll(leftRect.getNode(),**

```
        centerRect.getNode(), rightRect.getNode());
```

- Try running your application again - this time, the **`Rectangle`**s should all show up!



Creating the Bottom Pane

Let's create a **`labelPane`** with a label.

- Much like you did with **`createRectsPane()`**, write a method called **`createLabelPane()`** in the **`PaneOrganizer`** class that creates a **`Pane`**
    - Go to the **`BorderPane`** javadocs here and look for a method that will place the new pane at the bottom of the **`BorderPane`** (This method will place AND add the new pane). Hint: It's very similar to the **`setTop()`** method you used earlier...
    - Call this method in the constructor of your **`PaneOrganizer`**.

Adding contents to the Bottom Pane

- Back in **`createLabelPane()`**, declare and instantiate a **`Label`**. Pass a string in as an argument in its constructor. Add it to the **`labelPane`**'s list of children.
- Run your program. The label should show up, but it'll smashed against the left-most edge of the **`Pane`**. No good! We need to change the class of the Pane created in **`createLabelPane()`** to allow for specific layout capabilities.

- Go to the **`Pane`** class javadocs again and look at the subclasses. Find one that gives vertically stacked layout capabilities, and change the type of your **`labelPane`** to match.
- Use the javadocs to find a method allow you to position elements in your layout pane.
- Call the method you just found on **`labelPane`** and pass **`Pos.CENTER`** in as a parameter to center the items.
- Run your program - Everything should look fine visually

Checkpoint 2: Call a TA over to check your program!

- When you have finished this, switch roles with your partner: the navigator should now be the driver, and vice versa.

Part 2: Responding to User Input

Setting up an `EventHandler`

You've already seen **`EventHandler`**s in lecture-- if you add an **`EventHandler`** to a component like a **`Button`**, it will "listen" for **`Events`** (like button presses). Every time it detects an event, its **`handle`** method will execute. By writing your own **`EventHandler`** that implements this method, you can tell your program how to respond when the user presses the button.

Here's an example of a simple **`KeyEvent`** handler:

```
public class KeyHandler implements EventHandler<KeyEvent> {

	@Override
	public void handle(KeyEvent e) {
		KeyCode keyPressed = e.getCode();

		if (keyPressed == KeyCode.SPACE) {
			System.out.println("Spacebar!");
		}
}
```

In the code above the **`keyPressed`** variable stores what key was pressed on the keyboard. If the key used was the spacebar, "Spacebar!" will print in the terminal.

Now let's code it:

- We want to add a **`KeyHandler`** to our **`PaneOrganizer`** class - because it's specific to the **`PaneOrganizer`**, it can be a private class!
    - Why **`PaneOrganizer`** and not **`KeyableRect`**? If you want the key event to affect multiple **`KeyableRect`**'s, then would you want to put the **`KeyHandler`** in a place that is specific to only a single **`KeyableRect`** or to multiple **`KeyableRects`**? Where are multiple **`KeyableRects`** accessible?
    - Start by copying our **`KeyHandler`** code above into your **`PaneOrganizer.java`** file, making sure it's part of the **`PaneOrganizer`** class.
    - Change the visibility of the **`KeyHandler`** to **`private`**.

- Once again, you'll need to import several classes. If you try to run the program as is, the compiler won't know which which "**`EventHandler`**" or "**`KeyEvent`**" you want! Look up the specific names for the JavaFX versions of the classes you need, or let Eclipse help you out!

Adding an `EventHandler`

To add a **`KeyHandler`** to a component (let's say we have a **`Pane`** named **`myPane`**), we could say:

```
myPane.addEventHandler(KeyEvent.KEY_PRESSED, new
      KeyHandler());
```

The first argument to the **`addEventHandler`** method specifies what kinds of actions it should listen for. Other examples of **`KeyEvent`**s can be found **here**. Additionally, in lecture, we used **`setOnAction(EventHandler<ActionEvent>)`**. This method is only useful for buttons. To register other **`EventHandler`**s, use **`addEventHandler()`**. For more information, go here.

Now let's add the EventHandler using the code provided in the instructions above:

- We will want to add a new **`KeyHandler`** to one of the panes. We have 3 panes: _root, rectsPane, and labelPane. Which do you think would be best to add the **`KeyHandler`** to if we want to change the color of the **`KeyableRect`**?
- Run the app. Press the spacebar. Is "Spacebar!" printed in the console? It probably isn't. Let's find out why!
    - If it is, you should still read the bullets below, but probably don't need to act on them. Different implementations can behave a little differently.
    - In JavaFX, any graphical element that can be acted upon via user input (like a mouse or the keyboard) needs to belong to a "traversable cycle." That's just a fancy way of saying "all the stuff that JavaFX should check when a user enters something, to see if any of that stuff should be affected by the input". Since we haven't added our rectangle pane to this cycle yet, JavaFX doesn't know that your keyboard input should target that pane.
- How do we fix this? By setting the **`FocusTraversable`** property of our pane to **`true`**! Here is what that looks like in code, again assuming some example pane called **`myPane`**.

  myPane.setFocusTraversable(true);

  Complicated, huh?

- Now run the app again. If your **`EventHandler`**s are set up correctly, then every time you press the spacebar, you should see "Spacebar!" printed out to the console.

- One final order of business: we need to call **`event.consume()`** when dealing with **`Event`**s in our **`EventHandler`**s, because without it, the event will "travel" up the scene graph. For example, if a **`Pane`** contains a **`Button`**, and they *both* have an **`EventHandler`** for a mouse click installed, when the click is first registered on the button, it'll call the **`Button`**'s **`EventHandler`**'s **`handle()`** method, then the containing **`Pane`**'s **`EventHandler`**'s **`handle()`** method, and the **`EventHandler`** on the **`Pane`**'s parent, and so on. **As a rule of thumb: Always remember to `consume()` your `Event`s!** Do this now in our event handler!

Checkpoint 3: Switch roles with your partner: the navigator should now be the driver, and vice versa.

Next, we're going to modify the **`KeyHandler`** so that instead of just printing out "Spacebar!" it randomly changes the color of one of the **`Rectangle`**s

Helpful tip for the future:

`this` refers to the instance of the class that we are currently in—the class whose curly braces the **`this`** lives in. When a line of code is evaluated in a private inner class (which though private and inner, is indeed a class), **`this`** will refer to the instance of the private inner class. So in terms of our code, when inside **`KeyHandler`**'s curly braces, **`this`** refers to **`KeyHandler`**.

So when we want to refer to the outer class, we can tell the compiler to look at the outer class by prefacing our **`this`** with the name of the outer class. For example, saying **`PaneOrganizer.this`** inside an event handler would refer to the **`PaneOrganizer`** that contains the event handler!

Note: Only use the format **`<class name>.this`** when dealing with *outer classes from inner classes*. Otherwise, it is unnecessary and therefore bad style.

Customizing your EventHandler

- Write a method called **`changeColor()`** in your **`KeyableRect`** class to generate a random color and set the color of the **`Rectangle`** to that color.
    - The **`Color`** class has a static method called **`rgb()`** that takes in three **`int`**s, one for each of the red, blue, and green values that the new color should have, and returns an instance of the **`Color`** class. Make sure you're aware of the limited range these parameters can take. If you don't know what this means, head to the JavaDocs!!
    - To generate a random number, you'll want to use **`Math.random()`**. If you need a reminder on how to adjust this method's output for your desired range of values, consult the Graphics I lecture slides!

    - Use the **`setFill()`** method on the **`rectangle`** to change the color.

- Right now, our event handler just prints a message to the console. Let's give it some more meaningful functionality: changing the color of a **`KeyableRect`**. But how do we determine which rectangle we are changing the color of? Remember the **`keyPressed`** variable in the **`handle()`** method stores which key was used.
    - Implement a **`Switch`** based on this variable. The switch should have 3 cases, one for the L, C, and R keys on the keyboard which target the left, center, and right rectangles respectively. For each case, call your **`changeColor()`** method on the appropriate KeyableRect.

Run your program and check that everything is working correctly!

Finishing up: Adding quit functionality

- You're almost finished! But you need way to quit your program
- Add another case to the switch in your **`KeyHandler`** for when the user enters the "Q" button.
- Quit your app by calling **`Platform.exit();`**

Run your app and test it out!

Checkpoint 4: Call over a TA once everything is working in order to get checked off!