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 an **EventHandler** that allows 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 ([Graphics 1](#), [Graphics 2](#), [Graphics 3](#)).

You will be using several built-in classes from Java in this lab, so how will you find all the methods and properties available to use? That’s right, the javadocs! These javadocs were written by the Java developers and are frequently used by all coders! There is a convenient link to them on the CS15 website (Resources » Java Documentation >> [JavaFX-docs](#)), or choose the first result when you Google “JavaFX 8 API”. For this project, most of the classes you will be using are from javafx.scene.shape.

**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.

Up until this point, the majority of labs were partner coding. However, this will be a **solo** lab, so you will be working alone for the entirety of the assignment. Feel free to talk to your neighbors, but you must hand in your own code at the end.

As always, begin this lab by looking over the lab handout!

**Checkpoint 0:**
To do this lab, and all future projects in the course, you’ll be using Eclipse. Eclipse is a very popular Java “Integrated Development Environment” (IDE). An IDE is a software application that is designed to make writing code in a specific language easier for you. Up until now, you have been using Atom, which conveniently color-codes and auto-completes (somewhat), but it is limited other than that. Eclipse has a number of tools that will make your life much easier. These include:

- Automatic compiler that highlights syntax errors for you so you won’t have to worry about compile time errors anymore
- Code hinting, which gives you options to autocomplete what you are typing
- An auto-formatting function that automatically indents code when necessary
- Extremely time-saving automatic importing (great for JavaFX apps!)
- A run button that allows you to run code straight from the IDE
- A User Interface (UI) that is easier to use with big projects

Click [here](#) to access setup instructions (Complete this before you start the lab! Call over a TA to check that everything is set up correctly)
Part 1: Building a GUI with JavaFX Panes

Getting Started

First, you’ll be building the Graphical User Interface (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 four stencil files: `App.java`, `PaneOrganizer.java`, `KeyableRect.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 the method—by using the `Stage` class’ `setTitle(String s)` method.

- Try running the program (Remember you can run your code by right-clicking `App.java` in the `lab4` part of the Package Explorer and selecting Run As... >> Java Application). You’ll probably notice that your `Stage` does not appear. It turns out that a `Stage` won’t show up unless it’s told to be shown. Call the `show()` method on your `Stage` to ensure it shows up.

- Run the program again (You can re-run the last application you ran by clicking the green play button at the top of the screen). You should see a small `Stage` pop up in the top-left corner of your screen with a gray background. You can resize the `Stage` by clicking and dragging its bottom-right corner.

Adding Panes

Next, we want to add two `Panels` to our `Stage`: one on the top containing the rectangles, and one on the bottom with a label and quit button. These two `Panels` should be contained in a root `BorderPane`, which is the top-level organizational section we want our `Stage` to display. We will eventually add the `BorderPane` to our `Stage` by passing it into the `Scene` in our `App` class.
First, let's make the **BorderPane** in **PaneOrganizer** so that we can keep track of the two **Panes**.

- As mentioned above, we want to create a **BorderPane**, which 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 “_root” of type **BorderPane** in your **PaneOrganizer** class and instantiate it in the constructor. Hover over “BorderPane” to easily import the **BorderPane** class.

- Now write a method with the signature “**public BorderPane** **getRoot()**” that returns the **BorderPane** we created so that we can eventually add it to our **Stage**.

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 have just created to a new **Scene** by calling:

  ```java
  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**.
  - **TIP**: Try rewriting the code rather than copying it from the PDF to avoid errors.

- 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!

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 two Panes we discussed before: one contains the rectangles, and the other contains the label at the bottom of the app. Let’s first make the top Pane, which contains the rectangles.

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 width and height dimensions given in the Constants class. Remember how we can import Pane!
- Set the background color for the rectsPane in the same way you did for the BorderPane, but color it white this time (#FFFFFF).
- Add rectsPane to your BorderPane by using the setTop(...) method and passing in rectsPane.
  
  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: Look through the code you have written so far and make sure you understand what is happening. Having an understanding of using javafx is important for future projects, so if you have any questions call over a TA!

Adding Rectangles

Now that we know that our top Pane has been added, let’s add some Rectangles to it.

Go to the final class KeyableRect, which we already created for you with some stencil code included. The purpose of this class is to create a Rectangle shape with the capability to change color in response to a keyboard entry.
Overarching Design:
The only thing we can add to a Pane is a Node, so in order to add a KeyableRect, we need to create a getter method, getRectNode(), which would return KeyableRect's Rectangle node. In PaneOrganizer, you will call getRectNode() on a KeyableRect instance and add that directly to the Pane’s list of children.

In other words, when the PaneOrganizer wants to add a KeyableRect to its Pane, it will get the Node from KeyableRect and add it to its Pane.

Now let's code this design out:

- In the KeyableRect class, create an instance variable _rect of type Rectangle. Make sure to instantiate _rect in the constructor.

- Write a getter method called getRectNode() to return the rectangle Node from this KeyableRect. Hint: this method should return an object of type Node. Remember that practically all JavaFX objects are Nodes through inheritance. Polymorphism!!

Since KeyableRects can have different locations and colors, we can specify the location and color as arguments in the KeyableRect constructor!

- Modify the KeyableRect constructor’s parameters so that it takes in a double for the x location, a double for the y location, and a Color.

  Note: this color is a JavaFX class Color, which is different from the CSS color used earlier.

- Next, set the size of your Rectangle in the constructor using the width and height constants provided in the Constants class. Then, set its location and color using the values passed in from the constructor. Refer to the Graphics II lecture or JavaFX Shape Documentation for more information about how to accomplish this. (If looking at the Java Shape Documentation, look under javafx.scene.shape.Shape for information on setting the color).

Go back to the createRectsPane() method in the PaneOrganizer class.

- Declare and instantiate three KeyableRects called leftRect, centerRect, and rightRect. Set location (using the values we’ve provided in the Constants class) and color via the new KeyableRect’s constructor. To declare a color use the syntax Color.YELLOW or whichever color you want to use.
- If you run your program now, you will not see your KeyableRects, as they will not appear until they are added to rectsPane as children.
- To add them as children, call the line below after instantiating the three Rectangles:
  ```java
  rectsPane.getChildren().addAll(leftRect.getRectNode(),
                                 centerRect.getRectNode(), rightRect.getRectNode());
  ```
- Try running your application again - this time, the Rectangles should all show up!
Creating and Adding contents to the Bottom Pane

At this point, we've displayed the Rectangles in our application, but we haven't made them responsive to key input yet. In the meantime, we can now make the Pane with our label.

Let's create a labelPane with the label “Introduction to JavaFX!”.

- Similarly to createRectsPane(), write a private method called createLabelPane() in the PaneOrganizer class and call it in the PaneOrganizer constructor.

- In createLabelPane(), declare and instantiate a Label. Pass in a string as an argument in the Label's constructor. Then add it to the labelPane's list of children. Lastly, add labelPane to the BorderPane using setBottom() and passing in your instance of labelPane.

- Run your program. The label should show up, but it'll be 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.

- Change the type of your labelPane to be a VBox, which has vertically stacked layout capabilities.

- Call the setAlignment(...) method on labelPane and pass Pos.CENTER in as a parameter to center the label, and run your program.

Checkpoint 2: Call a TA over to check your program!
Part 2: Responding to User Input

Setting up an EventHandler

You've already seen EventHandlers 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 handle, you can tell your program how to respond when the user presses the button.

- In our PaneOrganizer class, the stencil code includes a KeyHandler, as we want the key event to affect multiple KeyableRects. Since it’s specific to the PaneOrganizer, it can be a private 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 “EventHandler” or “KeyEvent” you want! You can look up the specific names for the JavaFX versions of the classes you need, but using the Eclipse suggested imports is more efficient!

In the stencil code for the KeyHandler, the keyPressed variable stores what key was pressed. If the pressed key was the spacebar, “Spacebar!” will print in the terminal.

Adding an EventHandler

Let’s now add the event KeyHandler to the application so that it becomes responsive to user input. If we want the key event to affect multiple KeyableRect objects, should we add the KeyHandler to a place that is specific to only a single KeyableRect (the KeyableRect class) or a place that contains all the KeyableRects (the rectsPane in the PaneOrganizer)?

Note that a KeyHandler needs to be added to a JavaFX Pane, or else it will not register the key input. We will need to set the focus of this user input on the specific Pane. This is because any graphical element that can be affected by user input needs to belong to a “traversable cycle,” meaning that every JavaFX object should check when a user enters input. JavaFX will not know that the key input should target a Pane until it’s added it to this cycle by setting the focus on it.

In this case, we want to add the KeyHandler to the Pane containing all of the KeyableRects, i.e. the rectsPane, so that all three rectangles will be responsive to the key input.

Now let’s add the KeyHandler to rectsPane:
- Since we want to change the color of the KeyableRect, we should add the KeyHandler to rectsPane at the bottom of createRectsPane(), as shown below:
rectsPane.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 `KeyEvents` can be found [here](#). Additionally, in lecture, we used `setOnAction(EventHandler<ActionEvent>)`. This method is only useful for buttons. To register other `EventHandlers`, use `addEventHandler()`. For more information, go [here](#).

- Recall that we need to set the focus of the `KeyEvent` on the `rectsPane`. Since we haven’t added `rectsPane` to the traversable cycle yet, JavaFX doesn’t know that the key input should target it. To fix this, add the line below right after adding the `EventHandler`:
  ```java
  rectsPane.setFocusTraversable(true);
  ```

- Now run the app. If your `EventHandlers` are set up correctly, then every time you press the spacebar, you should see “Spacebar!” printed out to the console.

- Finally, you need to call `e.consume()` at the end of the `KeyHandler` to prevent the event from “traveling” up the scene graph.

**Checkpoint 3:** Take a moment to look over your program and make sure you understand the changes you just implemented.

### Customizing your EventHandler

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 `Rectangles`.

- Write a method called `changeColor()` in your `KeyableRect` class to generate a random color for the `Rectangle`.
  - The `Color` class has a static method `rgb()` that takes in three `ints` between 0 and 256—one for each of its red, green, and blue values—and returns an instance of the `Color` class.
  - To generate a random number, you’ll want to use `Math.random()`, which returns a double between 0 and 1. To adjust the output to be an integer between 0 and 256, you can multiply `Math.random()` by 256 and cast it as an integer. Consult the [Graphics I lecture slides](#) for further help if needed!
  - Use the `setFill()` method on the `rectangle` to change the color.

- Right now, `KeyHandler` just prints a message to the console. Let’s give it some more meaningful functionality: changing the color of a `KeyableRect`. To determine the specific rectangle whose color we are changing, we can refer to the key stored in the `keyPressed` variable in the `handle()` method.
  - Implement a `Switch` based on the `keyPressed` 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 first, you need a way to quit your program.
- Add another case to the switch in your `KeyListener` for when the user enters the “Q” key.
- Quit your app in this case 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!