Lecture 9

Intro to JavaFX

(photo courtesy of Snapchat filters)
This Lecture

• GUIs and JavaFX

• JavaFX Scene Graph Hierarchy
  o Aside: VBox

• Example: ColorChanger
  o Event Handling
  o Private inner classes
  o Random number generation
  o javafx.scene.paint.Colors

• Logical vs. Graphical Containment with JavaFX
  • Animation & Timelines

• Example: Clock
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
- Each graphical element positioned at specific pixel
What is JavaFX?

• Usually don’t want to program at the pixel level

• JavaFX is an API (Application Programming Interface) to a graphics+media library: a collection of useful classes, interfaces, and their methods (with suitable documentation) – no internals!

• Simply put, JavaFX is a set of graphics and media packages enabling developers to design, create, and test powerful graphical applications for desktop, web, and mobile devices

• JavaFX is a successor to the popular SWING library, previously used in CS15 projects to handle graphics!

• Shout out to 2016 CS15 TA Montana’s mom who worked on JavaFX!
Creating Applications from Scratch

• Until now, TAs took care of graphical components for you
  o Our support code defined the relevant classes
• From now on, you’re in charge of this!
• JavaFX is quite powerful but can be a bit tricky to wrap your head around because of the size of the JavaFX library
  o Not to fear, all JavaFX packages, classes, and method descriptions can be found in the JavaFX guide on our website!
Graphical User Interface (GUIs)

• GUIs provide a user-controlled (i.e., graphical) way to send messages to a system of objects, typically your app

• You’ll use JavaFX to create your own GUIs throughout the semester
Creating GUls With JavaFX (1/2)

- The main App class for our JavaFX application extends the abstract class javafx.application.Application.

- From now on in CS15, we’ll begin every project by implementing the Application class’ start method:
  - This method is called automatically by JavaFX to launch program.

- Java also creates javafx.stage.Stage that is passed into start method:
  - Once shown via the show method, the Stage is effectively a window for the application.
Creating GUls With JavaFX (2/2)

• In order for our application to be more than just a window, must first **set the scene** before **showing the stage**

• `javafx.scene.Scene` is the top-level container for all UI elements.
  - First instantiate `Scene` within `App` class’ `start` method
  - Then pass that `Scene` into `Stage`’s `setScene(Scene scene)` method to **set the scene**!

• In CS15, only specify 1 `Scene` – though JavaFX does permit creation of applications with multiple `Scenes`
  - Ex: A gaming application where you could select to play either DoodleJump, Tetris or Pacman from the main screen might utilize multiple `Scenes`- one for each subgame

• What exactly is a `javafx.scene.Scene`?
JavaFX Scene Graph Hierarchy (1/3)

• In JavaFX, the contents of the Scene (UI elements) are represented as a hierarchical tree, known as the Scene Graph

  o You are familiar with some other hierarchies already—containment and inheritance
JavaFX Scene Graph Hierarchy (2/3)

• You can think of the Scene Graph as a family tree of visual elements

• `javafx.scene.Node` is the superclass for all UI elements that can be added to the Scene, such as a Button or a Label

  o Each UI component that is added to the Scene Graph as a Node gets displayed graphically
JavaFX Scene Graph Hierarchy (3/3)

- Each Node can have multiple *children* and at most one *parent*
  - Child Nodes are almost always *graphically contained* in their parent Node
- The Node at the top of the Scene Graph is called the *root Node*
  - The root Node has no parent
The root of the Scene

- In CS15, root Node will be a `javafx.scene.layout.Pane` or one of its subclasses.
- Different Panes have different built-in layout capabilities to allow easy positioning of UI elements - see inheritance tree below for flavors.
- For now, use a VBox as the root of the Scene - more on VBox later.
Constructing the Scene Graph

- Instantiate root Node

```java
public class App extends Application {
    @Override
    public void start(Stage stage) {
        VBox root = new VBox();
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.show();
    }
}
```
Constructing the Scene Graph

• Instantiate root Node

• Then pass it into Scene constructor to construct Scene Graph
  
  o Scene Graph starts off as a single root Node with no children

```java
public class App extends Application {

    @Override
    public void start(Stage stage) {
        VBox root = new VBox();
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.show();
    }
}
```
Constructing the Scene Graph

• Once we set Scene and show Stage, we can begin populating Scene Graph

```java
public class App extends Application {

    @Override
    public void start(Stage stage) {
        VBox root = new VBox();
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.show();
    }
}
```
Adding UI elements to the Scene (1/2)

• How do we add more Nodes to the Scene Graph?

• Adding UI elements as children of root Node will add them to Scene and make them appear on the Stage!

• Calling getChildren() method on a Node returns a list of that Node’s children
  o By adding/removing Nodes from a Node’s list of children, we can add/remove Nodes from the Scene Graph!
Adding UI elements to the Scene (2/2)

• Recap: `getChildren()` returns a `List` of child Nodes
  
  o In example on right, `root.getChildren()` returns a list holding three `Buttons` (assuming we created them previously - next slide)

• To add a `Node` to this list of children, call `add(Node node)` on that returned list!
  
  o Can also use `addAll(Nodes... node1, node2, ...)` which takes in *any number of Nodes*
  
  o Allowing *any* number of arguments is a new capability of parameter list

• To remove a `Node` from this list of children, call `remove(Node node)` on that returned list!
root.getChildren().add(…) in action

• **Add 3 Buttons to the Scene by adding them as children of the root Node** (empty before this)

• **Order matters here!**

```java
/* Within App class */
@override
public void start(Stage stage) {
    // code for setting root, stage, scene elided
    Button b1 = new Button("Button 1");
    Button b2 = new Button("Button 2");
    Button b3 = new Button("Button 3");
    root.getChildren().addAll(b1, b2, b3);
}
```

*Remember double method call shorthand?*

`root.getChildren()` returns a list of `root`’s children. Rather than storing that returned list in a variable and calling `add(…)` on that variable, we can simplify our code by calling `add(…)` directly on the returned list of children!
Populating the Scene Graph (1/5)

• Similarly, can remove a UI element by removing it from root’s children
  
  o Note: The order of the children doesn’t matter when removing elements

/* Within App class */
@Override
public void start(Stage stage) {
  //code for setting root, stage, scene elided
  Button b1 = new Button("Button 1");
  Button b2 = new Button("Button 2");
  Button b3 = new Button("Button 3");
  root.getChildren().addAll(b1,b2,b3);
}
Populating the Scene Graph (2/5)

• Similarly, can remove a UI element by removing it from root’s children
  o Note: The order of the children doesn’t matter when removing elements

• Let’s remove third Button

/* Within App class */
@override
public void start(Stage stage) {
  //code for setting root, stage, scene elided

  Button b1 = new Button("Button 1");
  Button b2 = new Button("Button 2");
  Button b3 = new Button("Button 3");
  root.getChildren().addAll(b1, b2, b3);
  root.getChildren().remove(b3);
}

Populating the Scene Graph (3/5)

• What if we want to make more complex applications?

• Can add layout Pane as child of root Node, then add more UI elements as child Nodes of this Pane

• This will continue to populate the scene graph!
Populating the Scene Graph (4/5)

- First, instantiate a **VBox** and add as child of root **Node**

```java
/* Within App class */
@Override
public void start(Stage stage) {
    // code for setting scene elided

    Button b1 = new Button(); // no label
    Button b2 = new Button(); // no label
    root.getChildren().addAll(b1, b2);

    VBox holder = new VBox();
    root.getChildren().add(holder);
}
```
Populating the Scene Graph (5/5)

• Next, add Label to Scene as child of new VBox

/* Within App class */
@override
public void start(Stage stage) {
  //code for setting scene elided

  Button b1 = new Button();
  Button b2 = new Button();
  root.getChildren().addAll(b1,b2);
  VBox holder = new VBox();
  root.getChildren().add(holder);
  Label text = new Label(
      "I live in the VBox!" );
  holder.getChildren().add(text);
}

Note the default selection in blue
Removing a Node with children (1/2)

• Note that removing a Node with no children simply removes that Node...
  o root.getChildren().remove(b2); to remove second Button
Removing a **Node** with children (2/2)

- Note that while removing a **Node** with no children simply removes that **Node**...
  - `root.getChildren().remove(b2);` to remove second **Button**

- Removing a **Node** with children removes all of its children as well!
Removing a **Node** with children (2/2)

- Note that while removing a node with no children simply removes that node...
  - `root.getChildren().remove(b2);` to remove second **Button**

- Removing a **Node** with children removes all of its children as well!
  - `root.getChildren().remove(holder);` makes both **VBox** and **Label** disappear
Clicker Question

Given this code:

```java
public void start(Stage stage) {
    //code for setting scene elided

    Button b1 = new Button();
    Button b2 = new Button();
    root.getChildren().addAll(b1, b2);

    VBox holder = new VBox();
    root.getChildren().add(holder);
    Label removeLabel = new Label("remove me!");
    holder.getChildren().add(removeLabel);
}
```

Which of the following correctly removes `removeLabel` from the VBox `holder`?

A. `root.remove(removeLabel);`
B. `holder.remove(removeLabel);`
C. `root.getChildren().remove(removeLabel);`
D. `holder.getChildren().remove(removeLabel);`
VBox layout pane (1/5)***

• So what exactly is a VBox?
• VBox layout Pane creates an easy way for arranging a series of children in a single vertical column
• We can customize vertical spacing between children using VBox’s setSpacing(double) method
  o the larger the double passed in, the more space between the child UI elements
VBox layout pane (2/5)**

• Can also set positioning of entire vertical column of children

• Default positioning for the vertical column is in TOP_LEFT of VBox (Top Vertically, Left Horizontally)
  
  o can change Vertical/Horizontal positioning of column using VBox’s `setAlignment(Pos position)` method, passing in a `javafx.geometry.Pos` constant

• Pos options are in the form Pos.VERTICAL_HORIZONTAL
  
  o e.g. `Pos.BOTTOM_RIGHT` represents positioning on the bottom vertically, right horizontally
  
  o full list of Pos constants can be found here

Why ALL_CAPS `notation`?
It is a “symbolic constant” with pre-defined meaning.
VBox layout pane (3/5)**

- The following code produces the example on the right:

```java
VBox root = new VBox();

Button b1 = new Button("Top");
Button b2 = new Button("Middle");
Button b3 = new Button("Bottom");
root.getChildren().addAll(b1, b2, b3);

Scene scene = new Scene(root, 200, 200);
stage.setTitle("Sample VBox");
stage.setScene(scene);
stage.show();
```
VBox layout pane (4/5)***

- Adding spacing between children

VBox root = new VBox();

Button b1 = new Button("Top");
Button b2 = new Button("Middle");
Button b3 = new Button("Bottom");
root.getChildren().addAll(b1,b2,b3);

root.setSpacing(8);

//code for setting the Scene elided
VBox layout pane (5/5)**

• Setting alignment property to configure children in TOP (vertically) CENTER (horizontally) of the VBox

```java
VBox root = new VBox();

Button b1 = new Button("Top");
Button b2 = new Button("Middle");
Button b3 = new Button("Bottom");
root.getChildren().addAll(b1, b2, b3);

root.setSpacing(8);
root.setAlignment(Pos.TOP_CENTER);

// code for setting the Scene elided
```
CS15 PaneOrganizer Class (1/2)

• Until now, all code dealing with the Scene has been inside Application’s start method; adding more widgets will clutter it up…

• In CS15, write a PaneOrganizer class where all graphical application logic will live – an example of “delegation” pattern

• Delegation removes application-dependent code from App class, which only creates scene and instantiates a PaneOrganizer

• PaneOrganizer will instantiate root Pane, and provide a publicgetRoot() method that returns this root
  
  o App class can now access root Pane through PaneOrganizer’s publicgetRoot() method and pass root into Scene constructor

• We’ll do this together in a sec!
Pattern

1. App class instantiates a PaneOrganizer which creates root, then passes return value from getRoot() to Scene constructor, so Scene has a root

2. Top-level PaneOrganizer class instantiates JavaFX UI components (Button, Label, Pane...), then adds them to root Pane (and the Scene, indirectly) using root.getChildren().add(...); or root.getChildren().addAll(...);
Our First JavaFX Application: ColorChanger

- Spec: App that contains text reading “CS15 Rocks!” and a Button that randomly changes text’s color with every click
- Useful classes: Stage, Scene, VBox, Label, Button, EventHandler
Process: **ColorChanger**

1. Create an `App` class that extends `javafx.application.Application` and implements `start` (where you set Scene)

2. Create a `PaneOrganizer` class that instantiates root `Pane` and provides a public `getRoot()` method to return the Pane. In `PaneOrganizer`, instantiate a `Label` and a `Button` and add them as children of root `Pane`.

3. Set up a custom `EventHandler` that changes `Label`'s color each time `Button` is clicked, and register `Button` with this new `ClickHandler`.
ColorChanger: App class

1. When implementing start, we:

   A. 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();
        /*write our PaneOrganizer class later,
         where we will instantiate the root Pane */
    }
}
```
ColorChanger: App class

1. When implementing `start`, we:

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

   B. Instantiate a new `Scene`, passing in:
      - root `Pane`, accessed through `organizer`'s public `getRoot()`
      - along with desired width and height of `Scene`

```java
public class App extends Application {
    @Override
    public void start(Stage stage) {
        PaneOrganizer organizer = new PaneOrganizer();
        /*write our PaneOrganizer class later, where we will instantiate the root Pane */
        Scene scene =
        new Scene(organizer.getRoot(), 80, 80);
        stage.set(scene);
        stage.setTitle("Color Changer!");
        stage.show();
    }
}
```
ColorChanger: App class

1. When implementing **start**, we:

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

   B. Instantiate a new **Scene**, passing in:
      - the root **Pane**, which is accessed through **organizer**'s public **getRoot()** method
      - along with desired width and height of **Scene**

   C. Set the **Scene**, title the **Stage**, and show the **Stage**

```java
public class App extends Application {

    @Override
    public void start(Stage stage) {
        PaneOrganizer organizer = new PaneOrganizer();
        /*write our PaneOrganizer class later,
        where we will instantiate the root Pane */

        Scene scene =
            new Scene(organizer.getRoot(), 80, 80);
        stage.setScene(scene);
        stage.setTitle("Color Changer!");
        stage.show();
    }
}
```
Process: **ColorChanger**

1. Create App class that extends `javafx.application.Application` and implements `start` (where you set Scene!)

2. Create `PaneOrganizer` class that instantiates root `Pane` and provides public `getRoot()` method to return the `Pane`. In `PaneOrganizer`, instantiate a `Label` and `Button` and add them as children of root `Pane`

3. Set up a custom `EventHandler` that changes `Label`'s color each time `Button` is clicked, and register `Button` with this new `ClickHandler`
ColorChanger: Our PaneOrganizer Class

2. In writing our PaneOrganizer class, we

A. Instantiate root VBox and store it in instance variable _root
**ColorChanger: Our PaneOrganizer Class**

2. In writing our PaneOrganizer class, we

   A. Instantiate root VBox and store it in instance variable _root

   B. Create a publicgetRoot() method that returns _root
      - Reminder: this makes root Pane accessible from within App class

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

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

    public VBox getRoot() {
        return _root;
    }
}
```
2. In writing our **PaneOrganizer** class, we

C. Instantiate **Label** and **Button**, passing in **String** representations of text we want displayed

- **_label** is an instance variable because need to access it elsewhere to change its color
- **btn** is a local variable because only need to access it from within constructor

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

    public PaneOrganizer() {
        _root = new VBox();
        _label = new Label("CS15 Rocks!");
        Button btn = new Button("Random Color");
    }

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

2. In writing our PaneOrganizer class, we

C. Instantiate Label and Button, passing in String representations of text we want displayed
   - label is an instance variable because need to access it elsewhere to change its color
   - btn is a local variable because only need to access it from within constructor

D. Add Label and Button as children of root
   - root.setSpacing(8) is optional but creates a nice vertical distance between Label and Button

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

    public PaneOrganizer() {
        _root = new VBox();
        _label = new Label("CS15 Rocks!");
        Button btn = new Button("Random Color");
        _root.getChildren().addAll(_label,btn);
        _root.setSpacing(8);
    }

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

1. Create **App** class that extends `javafx.application.Application` and implements `start` (where you set Scene!)

2. Create **PaneOrganizer** class that instantiates root **Pane** and provides public `getRoot()` method to return the **Pane**. In **PaneOrganizer**, instantiate a **Label** and **Button** and add them as children of root **Pane**

3. Set up a custom **EventHandler** that changes **Label**’s color each time **Button** is clicked, and register **Button** with this new **ClickHandler**
Responding to User Input

• Need a way to respond to stimulus of **Button** being clicked

• We refer to this as **Event Handling**
  
  o A source (**Node**), such as a **Button**, generates an **Event** (such as a mouse click) and notifies all registered **EventHandler**s

  o **EventHandler** is an interface, so all classes that implement **EventHandler** must implement **handle(Event event)** method, which defines response to event
EventHandler (1/3)

- **Button** click causes JavaFX to generate a `javafx.event.ActionEvent`
  - `ActionEvent` is only one of many JavaFX `eventType`s that are subclasses of `Event`
- Classes that implement `EventHandler` can polymorphically handle any subclass of `Event`
  - When a class implements `EventHandler` interface, it must specify what type of `Event` it should know how to handle
  - How do we do this?
- `EventHandler` interface declared as: `public interface EventHandler<T extends Event>...`
  - The code inside `< >` is known as a “generic parameter”—more on these later!
  - Lets you specialize the interface to deal in all its methods only with a specialized subclass of `Event`
  - Forces you to replace what is inside the literal `< >` with some subclass of `Event`, such as `ActionEvent`, whenever you write a class that implements `EventHandler` interface (see next slide)
**EventHandlers (2/3)**

- We want our custom `EventHandler` (call it `ClickHandler`) to handle an `ActionEvent`, so `ClickHandler` will implement "`EventHandler<ActionEvent>`” interface
  - literally, “< >” included!!

```java
public class ClickHandler implements EventHandler<ActionEvent> {
    public void handle(ActionEvent e) {
        //code to change _cs15Rocks_ Label
    }
}
```
**EventHandlers (3/3)**

- Our ClickHandler must implement the `handle(ActionEvent e)` method of the `EventHandler` interface, which will specify the response to the `ActionEvent` (in this case, a click)
  - For now, you most likely won’t need to use the parameter `e`
- To tell this new ClickHandler to *listen* for the Button’s `ActionEvent`, register `Button` with the ClickHandler by calling `btn.setOnAction`, passing in an instance of our ClickHandler class
  - The mechanics of handing off the event to the handler happen under the hood of JavaFX

```java
public class ClickHandler implements EventHandler<ActionEvent> {
    public void handle(ActionEvent e) {
        //code to change _cs15Rocks Label
    }
}

public class PaneOrganizer {
    //instance variable declarations elided
    public PaneOrganizer() {
        _root = new VBox();
        _label = new Label("CS15 Rocks!");
        Button btn = new Button("Random Color");
        _root.getChildren().addAll(_label, btn);
        _root.setSpacing(8);
        btn.setOnAction(new ClickHandler());
    }
    //code to return root elided
}
```
1. Create an App class that extends `javafx.application.Application` and implements `start` (where you set `Scene`!)

2. Create a `PaneOrganizer` class that instantiates root `Pane` and provides a public `getRoot()` method to return the `Pane`. In `PaneOrganizer`, instantiate a `Label` and a `Button` and add them as children of root `Pane`.

3. Define a custom `EventHandler` that changes `Label`’s color each time `Button` is clicked, and register `Button` with this new `ClickHandler`.

Back to **ColorChanger**: Process
ColorChanger: ClickHandler class

3. Defining our custom EventHandler, ClickHandler

- **ClickHandler** must listen for click event and respond to it by changing the color of “CS15 Rocks!” Label

- How will **ClickHandler** access Label?
  - Multiple ways to do this: could have **ClickHandler** constructor take in a Label as a parameter
  - This works, but is there a better way?

```java
public class ClickHandler implements EventHandler<ActionEvent> {
    private Label _cs15Rocks;
    public ClickHandler(Label label) {
        _cs15Rocks = label;
    }
    public void handle(ActionEvent e) {
        // code to change _cs15Rocks Label
    }
}
```
Aside: Private Inner Classes (1/2)

• Until now, all classes we have created have been public
  - live in their own file
  - can be accessed from within any class

• Now we’ll introduce private inner classes!
  - useful when there is a class, such as an `EventHandler`, for which only need to create a single instance, from within a single class
  - Private inner classes have access to the instance variables/methods of class they are contained in (that declared them)
  - Inner classes are a convenient and safe shortcut- don’t require a file
Aside: Private Inner Classes (2/2)

• Rather than making the `ClickHandler` class a public class in its own file, we can make it a private inner class of the `PaneOrganizer` class

• Our `ClickHandler` will then have access to `PaneOrganizer`’s `_label` instance variable

• Can then set `_label` ’s text color from within `ClickHandler`’s `handle(ActionEvent)` method, without needing to deal with any unnecessary passing around of references to `Label`
3. **Defining our custom EventHandler, ClickHandler:**

- In order to make ClickHandler a private inner class of PaneOrganizer class, we simply declare ClickHandler as a private class and place it within brackets of public PaneOrganizer class.
3. Defining our custom EventHandler, ClickHandler:

- Now must implement handle method

- How will ClickHandler generate a random color whenever btn’s ActionEvent is detected?

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

    public PaneOrganizer() {
        _root = new VBox();
        _label = new Label("CS15 Rocks!");
        Button btn = new Button("Random Color");
        _root.getChildren().addAll(_label, btn);
        _root.setSpacing(8);
    }

    public VBox getRoot() {
        return _root;
    }

    private class ClickHandler implements EventHandler<ActionEvent> {
        public ClickHandler() { //code elided
            public void handle(ActionEvent event) {
                //implementation elided for now
            }
        }
    }
}
```
Generating `javafx.scene.paint.Color`

- We can generate most colors of visible color spectrum by additive mixtures of Red, Green and Blue “primaries” generated by display hardware
  - each display pixel has an R,G, and B sub-pixel to do this color mixing

*diagram of color spectrum*

- `javafx.scene.paint.Color` has static method `rgb(int red, int green, int blue)` that returns a custom color according to specific passed-in Red, Green, and Blue integer values in [0-255]
  - ex: `Color.WHITE` can be expressed as `Color.rgb(255,255,255);`
ColorChanger: Our EventHandler, ClickHandler

3. Defining our custom EventHandler, ClickHandler:

- **Math.random()** returns a random double between 0 inclusive and 1 exclusive

- Multiplying this value by 256 turns [0, 1) double into a [0, 256) double, which we cast to a [0,255] int by using new (int) cast operator

- Use these ints as Red, Green, and Blue RGB values for a custom javafx.scene.paint.Color

- Call setTextFill on _label, passing in new random Color we’ve created

```java
private class ClickHandler implements EventHandler<ActionEvent> {
    public ClickHandler() {
        //code elided
    }
    public void handle(ActionEvent event) {
        int red = (int) (Math.random()*256);
        int green = (int) (Math.random()*256);
        int blue = (int) (Math.random()*256);
        Color customColor = Color.rgb(red,green,blue);
        _label.setTextFill(customColor);
    }
}
```
3. **Defining our custom EventHandler, ClickHandler:**

   - **Last step is to register the Button with the click Event**
   
   - **To do so, call setOnAction on btn, passing in an instance of our ClickHandler**

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

    public PaneOrganizer() {
        _root = new VBox();
        _label = new Label("CS15 Rocks!");
        Button btn = new Button("Random Color");
        _root.getChildren().addAll(_label, btn);
        _root.setSpacing(8);
        btn.setOnAction(new ClickHandler());
    }

    public VBox getRoot() {
        return _root;
    }

    private class ClickHandler implements EventHandler<ActionEvent> {
        // code on previous slide
    }
}
```
The Whole App: 

**ColorChanger**

```java
public class App extends Application {
    @Override
    public void start(Stage stage) {
        PaneOrganizer organizer = new PaneOrganizer();
        Scene scene = new Scene(organizer.getRoot(), 80, 80);
        stage.setScene(scene);
        stage.setTitle("Color Changer");
        stage.show();
    }
}
```

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

    public PaneOrganizer() {
        _root = new VBox();
        _label = new Label("CS15 Rocks!");
        Button btn = new Button("Random Color");
        _root.getChildren().addAll(_label, btn);
        _root.setSpacing(8);
        btn.setOnAction(new ClickHandler());
    }

    public VBox getRoot() {
        return _root;
    }
}
```

```java
private class ClickHandler implements EventHandler<ActionEvent> {
    //constructor elided
    public void handle(ActionEvent event) {
        int red = (int) (Math.random() * 256);
        int green = (int) (Math.random() * 256);
        int blue = (int) (Math.random() * 256);
        Color customColor = Color.rgb(red, green, blue);
        _label.setTextFill(customColor);
    }
}
```
Putting It All Together
Logical vs. Graphical Containment/Scene Graph

- Graphically, **VBox** is a pane contained within **Scene**, but logically, i.e., programmatically, **VBox** is contained within **PaneOrganizer**

- Graphically, **Button** and **Label** are contained within **VBox**, but logically, **Button** and **Label** are contained within **PaneOrganizer**

- Logical containment is based on where objects are instantiated, while graphical is based on JavaFX elements being added to other JavaFX elements via `getChildren.add(…)` method
Announcements (1/2)

• TASafeHouse deadlines have changed
  o Early handin is **Friday 10:00pm**
  o On time handin is **Sunday 11:59pm**

• Please spend some time reviewing these slides on your own to make sure you fully understand them
  o JavaFX is a dense topic that will be *essential* to all future assignments

• We have posted a [JavaFX guide](#) to our website
  o Read this if you’d like a more complete overview of JavaFX and extra examples
  o Note: a couple examples may seem confusing, but don’t worry, any unfamiliar syntax will all be covered in lecture by next Thursday!
Announcements (2/2)

• Lab 3 goes out today!
  ○ You must get Lab 2 checked off by the end of your lab this week to get credit

• Review Sessions are this Thursday and Sunday