Lecture 8

Graphics Part I
Intro to JavaFX

(photography courtesy of Instagram filters)
Initial Survey Responses

How much programming experience do you have?
359 responses

- None: 46.5%
- Some: 50.1%
- A lot: 3.3%

12 responses (3.3%) indicated having a lot of programming experience.
Initial Survey Responses

In which programming languages do you feel comfortable programming, if any?

351 responses

- None: 204 (58.1%)
- Java: 78 (22.2%)
- C/C++: 19 (5.4%)
- HTML/CSS: 52 (14.8%)
- Javascript: 36 (10.3%)
- Python/Ruby: 81 (23.1%)
- Matlab: 4 (1.1%)
- Racket: 2 (0.6%)
- Pyret: 1 (0.3%)
- R, Stata: 1 (0.3%)
- C#: 1 (0.3%)
- Java, a little bit: 1 (0.3%)
- PHP: 1 (0.3%)
- Python: 1 (0.3%)
- only very basic Python: 1 (0.3%)
Initial Survey Responses

How familiar were you with object-oriented concepts (before starting the course)?

360 responses

- 55.6%: Not at all but excited to learn!
- 37.2%: I have heard a few terms...
- 7.2%: Super duper familiar, I love objects
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**
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 is positioned at specific pixel
What is JavaFX?

• Usually don’t want to program at the pixel level – far too tedious!

• 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 an API (Application Programming Interface) to a graphics and media library: a collection of useful classes and interfaces and their methods (with suitable documentation) – no internals accessible!
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 are 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 user-controlled (i.e., graphical) way to send messages to a system of instances, typically your app.

- Use JavaFX to create your own GUIs throughout the semester.
Components of JavaFX application

• Stage
  o location (or “window”) where all graphic elements will be displayed

• Scene
  o container for all UI elements to be displayed on a stage (blue border with “Stage” label)
    ▪ UI elements include Panes, Labels, Shapes, etc. Like the Button shown
  o scene must be on a stage to be visible (grey interior portion)

• Scene Graph
  o family tree of graphical elements

• Nodes
  o all elements of the Scene Graph
  o graphical representation called a UI element, widget, or control (synonyms)
Creating GUIs With JavaFX: Stage (1/2)

- **App** class for JavaFX application extends imported **abstract** class `javafx.application.Application`
  
- From now on, begin every project by implementing **Application**'s **abstract** `start()`  
  
  - `start()` is called automatically by JavaFX to launch program

- Java automatically creates a **Stage** using the imported `javafx.stage.Stage` class, which is passed into `start()`  
  
  - when `start()` calls stage's `show()`, stage becomes a window for the application
Creating GUIs With JavaFX: Scene (2/2)

- In order for our application to provide **content** for what to show on the stage, must first **set (specify) a scene** before **showing it on (in) 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

- So what exactly is a **`javafx.scene.Scene`**?
JavaFX Scene Graph Hierarchy (1/3)

• In JavaFX, contents of the Scene (UI elements) are represented as a hierarchical tree, known as the Scene Graph
  - you are familiar with some other hierarchies already - containment and inheritance

containment hierarchy

inheritance hierarchy

directed graph
JavaFX Scene Graph Hierarchy: Nodes (2/3)

• Think of the Scene Graph as a *family tree of visual elements*

• `javafx.scene.Node` is the abstract superclass for all UI elements that can be added to the `Scene`, such as a `Button` or a `Label`
  
  o all UI elements are concrete subclasses of `Node` (`Button`, `Label`, `Pane`, etc.)

• Each UI component that is added to the Scene Graph as a `Node` gets displayed *graphically*
JavaFX Scene Graph Hierarchy: Node Properties (3/3)

• Each Node can have multiple children but at most one parent
  o child Nodes are almost always graphically contained in their parent Node
  o more on graphical containment later!

• The Node at the top of the Scene Graph is called the root Node
  o the root Node has no parent
The root of the **Scene**

- Root **Node** will **always** be a `javafx.scene.layout.Pane` or one of `Pane`'s subclasses.
- Different **Panels** 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 (1/3)

• Instantiate root Node

```java
public class App extends Application {
    @Override
    public void start(Stage stage) {
        // code to populate Scene Graph
        VBox root = new VBox();
    }
}
```
Constructing the Scene Graph (2/3)

• Instantiate root Node

• Pass it into Scene constructor to construct Scene Graph
  o Scene Graph starts off as a single root Node with no children
  o the root is simply a container, without graphical shape

```java
public class App extends Application {
    @Override
    public void start(Stage stage) {
        // code to populate Scene Graph
        VBox root = new VBox();
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.show();
    }
}
```
Constructing the Scene Graph (3/3)

Once we `setScene()` and `show()` on `Stage`, we begin populating the Scene Graph.

```java
public class App extends Application {
    @Override
    public void start(Stage stage) {
        // code to populate Scene Graph
        VBox root = new VBox();
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.show();
    }
}
```
Adding UI Elements to the Scene (1/2)

• How to add more **Nodes** to the Scene Graph?

• Adding UI elements as **children** of root **Node** adds them to **Scene** and makes them appear on **Stage**!

• Calling **getChildren()** method on a **Node** returns a list of that **Node**’s children
  - by adding/removing **Nodes** from a **Node**’s list of children, we can add/remove **Nodes** from the Scene Graph!
  - later we’ll see how Java supports Lists as a type
Adding UI Elements to the Scene (2/2)

• `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 lists

• 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 (no children before this)

• First create buttons

• Then add buttons to Scene Graph

/* 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);
}

Note the default button selection in blue

Remember double dot 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!
Removing UI Elements from the Scene

• Similarly, remove a UI element by removing it from the list of its parent’s children.
  o note: order of children doesn’t matter when removing elements since you specify their variable names

• Let’s remove third Button*

/* Within App class */
@Override
go 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);
}

*Note: not a typical design choice to add and then remove a Node in the same code block!
Populating the Scene Graph (1/3)

• What if we want to make more complex applications?
• Add specialized layout containers, called Panes
• Add another 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 (2/3)

- First, instantiate another VBox and add it as child of root Node
  - **Note**: VBox is a pure container without graphical shape

```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 (3/3)

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

```java
/* 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);
}
```
Removing a **Node** with children (1/3)

• Removing a **Node** with no children simply removes that **Node**…
  - `root.getChildren().remove(b2);` to remove second **Button**
Removing a Node with children (2/3)

• 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 removes all of its children as well!
Removing a **Node** with children (3/3)

- Note that 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 its **Label** disappear
Lecture Question

Given this code:

```java
public void start(Stage stage) {
    // code for setting scene elided
    // parallel code for setting up root 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 would next remove `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)
  
  - can change Vertical/Horizontal positioning of column using VBox’s `setAlignment(Pos position)` method, passing in a `javafx.geometry.Pos` constant—`javafx.geometry.Pos` is a class of enums, or fixed set of values, to describe vertical and horizontal positioning. Use these values just like a constants class that you would write yourself!

- Pos options are in the form Pos.<vertical position>_<horizontal position>
  
  - e.g. `Pos.BOTTOM_RIGHT` represents positioning on the bottom vertically, right horizontally
  
  - full list of Pos constants can be found [here](#)

---

*Why ALL_CAPS notation?*

It is a “symbolic constant” with pre-defined meaning.
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();
```

Overloaded `Scene` constructor with three parameters: other `Scene` constructors have different parameter lists.
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 nodes will clutter it up…

- 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 – another example of “divide et impera”

- PaneOrganizer will instantiate root Pane, and provide a public `getRoot()` method that returns this root
  - App class can now access root Pane through PaneOrganizer’s public `getRoot()` method and pass root into Scene constructor

- We’ll do this together soon!
CS15 PaneOrganizer Class (2/2)

Pattern

1. **App** class instantiates a **PaneOrganizer**, which creates root

2. **App** class passes return value from `getRoot()` to **Scene** constructor, so **Scene** has a root

3. Top-level **PaneOrganizer** class instantiates JavaFX UI components (**Button**, **Label**, **Pane**...)

4. These UI components are added to root **Pane** (and therefore to 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 **App** class that extends `javafx.application.Application` and implements `start` (where you set `Scene`) – the standard pattern

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`
1. To implement **start**: 

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 (2/3)

1. To implement **start**:
   
   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();
     }
   }
   ```

   root width height

1. To implement `start`:

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()` 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`.
2. To write PaneOrganizer class:

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

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

    public PaneOrganizer() {
        _root = new VBox();
    }
}
```
ColorChanger: Our PaneOrganizer Class (2/4)

2. To write PaneOrganizer class:

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

B. Create a public getRoot() method that returns _root

  o reminder: this makes root Pane accessible from within App’s start for new Scene(root)

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

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

        Button btn = new Button("Random Color");
        _root.getChildren().addAll(_label, btn);
        _root.setSpacing(8);

        btn.setOnAction(new ClickHandler());
    }

    public VBox getRoot() {
        return _root;
    }
}
```
2. To write PaneOrganizer class:

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 in P.O. 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 (4/4)**

2. To write `PaneOrganizer` class:

   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 in P.O. 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 instances of EventHandler

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

  o note that handle(Event event) is called by JavaFX, not the programmer
EventHandlers (1/3)

• Button click causes JavaFX to generate a `javafx.event.ActionEvent`
  o `ActionEvent` is only one of many JavaFX `EventTypes` that are subclasses of `Event` class

• Classes that implement `EventHandler` interface can polymorphically handle any subclass of `Event`
  o when a class implements `EventHandler` interface, it must specify what type of `Event` it should know how to handle
  o how do we do this?

• `EventHandler` interface declared as: `public interface EventHandler<T extends Event>...`
  o the code inside literal `< >` is known as a “generic parameter” – this is magic for now
  o lets you specialize the interface to deal in all its methods only with a specialized subclass of `Event`
  o 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
EventHandlers (2/3)

• We can create an EventHandler and call it ClickHandler

• This EventHandler will handle an ActionEvent, meaning that ClickHandler will implement the “EventHandler<ActionEvent>” interface
  
    o 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 on a button)
  o 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
  o the mechanics of handing off the event to the handler happen under hood of JavaFX

```java
public class ClickHandler implements EventHandler<ActionEvent> {
    public ClickHandler() { //code elided }

    @Override
    public void handle(ActionEvent e) {
        //code to change _label
    }
}
```

```java
//elsewhere in program
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());
    }
}
```

Back to 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. Define a custom `EventHandler` that changes `Label`’s color each time `Button` is clicked, and register `Button` with this new `ClickHandler`.
### 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 _myLabel;
    public ClickHandler(Label label) {
        _myLabel = label;
    }
    @Override
    public void handle(ActionEvent e) {
        // code to change _myLabel Label
    }
}
```
Aside: Private Inner Classes (1/2)

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

• Introducing private inner classes!
  o useful when there is a class, such as an EventHandler, for which you only need to create a single instance, from within a single class
  o private inner classes have access to instance variables/methods of the class that contains them (that declared them)
  o 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 braces of public `PaneOrganizer` class.

```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> {
    }
}
```
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() { // simple 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 a R,G, and B sub-pixels to do this color mixing

![Color Palette](image)

- `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 (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
    }
    @Override
    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);
    }
}
```
ColorChanger: Back to our PaneOrganizer Class

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 (Did this on Slide 52)

```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
    }
}
```
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();
}
}

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> {
//constructor elided

@Override
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);
}
}

public class PaneOrganizer {
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.application.*;

// PaneOrganizer imports
// package includes Pane class and its subclasses
import javafx.scene.layout.*;
//package includes Label, Button classes
import javafx.scene.control.*;
//package includes ActionEvent, EventHandler classes
import javafx.event.*;
import javafx.scene.paint.Color;

public class App extends Application {

//App class imports
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.application.*;

// PaneOrganizer imports
// package includes Pane class and its subclasses
import javafx.scene.layout.*;
//package includes Label, Button classes
import javafx.scene.control.*;
//package includes ActionEvent, EventHandler classes
import javafx.event.*;
import javafx.scene.paint.Color;

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> {
//constructor elided

@Override
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**, VBox is contained within PaneOrganizer.

- **Graphically**, Button and Label are contained within VBox, but **logically**, Button and Label are contained within PaneOrganizer, which has no graphical appearance.

- **Logical** containment is based on where instances are instantiated, while **graphical** containment is based on JavaFX elements being added to other JavaFX elements via `getChildren.add(…)` method, and on the resulting scene graph.
Announcements

• Fruit Ninja deadlines:
  - Early: Friday, 2/19 at 11:59pm
  - On-time: Sunday, 2/21 at 11:59pm
  - Late: Tuesday, 2/23 at 11:59pm

• Sections will be a Design Discussion this week!

• Mentorship form is out! Please fill this out by tomorrow at midnight!

• Please spend some time reviewing these slides on your own to make sure you fully understand them
  - JavaFX is a dense topic that will be essential to all future assignments
  - There is a JavaFX guide on the website!
    - All remaining material will be covered by Graphics II and III
    - You get to exercise your creativity in making a simple interactive cartoon of your own design😊

Andries van Dam © 2021 02/18/21
IT in the News

ft. Socially Responsible Computing!
Algorithms & Decision Making: Vaccines (1/2)

- **December 2020**: Stanford used algorithm to allocate vaccines – left out medical residents who treat COVID patients

How did this happen?

- Use of allocation algorithm based on inaccurate assumptions
  - Weighted age highly, despite older clinicians using telemedicine
  - Based on pre-pandemic job responsibilities: not reflective of current situation
- **Lack of thorough critical thinking**

“[A]lgorithms are made by people and the results … were reviewed multiple times by people. The ones who ultimately approved the decisions are responsible.”

– Stanford medical resident
Algorithms & Decision Making: Medical Care (2/2)

- **Larger problem:** bias in the way US healthcare is allocated
  - documented racial bias, gender bias, fatphobia…
- **Proposed solution:** use algorithms based on existing data!
  - → Increasing use of AI/ML to allocate care, assign “risk scores”
- **But…**
  - Proven **bias against Black patients** → worse health outcomes
  - Patients are unaware, do not consent to use of algorithms

**How did this happen?**
- Profit-based healthcare → need to cut costs: algorithms cheaper than humans
- Naïve trust in algorithms and data to be “neutral” or “unbiased”

**Takeaways:**
- Data-based algorithms **reproduce existing bias**, rather than reducing it.
- Trust in neutrality of algorithms → difficult to challenge biased decisions.