Lecture 9

Graphics Part I

Intro to JavaFX

(photo courtesy of Instagram filters)
Outline

- GUIs and JavaFX
- JavaFX Scene Graph Hierarchy
- VBox panes and PaneOrganizers
- Example: ColorChanger
- Event Handling and lambda expressions
- 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:
  o origin in upper-left corner
  o x-axis increasing left to right
  o y-axis increasing top to bottom
  o 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
  - location (or “window”) where all graphic elements will be displayed

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

- Scene Graph
  - family tree of graphical elements

- Nodes
  - all elements of the Scene Graph
  - 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

```java
public class App extends Application {
    //mainline provided by TAs elided
    @Override
    public void start(Stage stage) {
        stage.show();
    }
}
```
Creating GUIs With JavaFX: Scene (2/2)

• 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
  o first instantiate `Scene` within `App` class’ `start` method
  o 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`
  o ex: an arcade 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`?
Outline

- GUIs and JavaFX
- JavaFX Scene Graph Hierarchy
- VBox panes and PaneOrganizers
- Example: ColorChanger
- Event Handling and lambda expressions
- Logical vs. Graphical Containment with JavaFX
JavaFX Scene Graph Hierarchy

- 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/association** and **inheritance/interface**
JavaFX Scene Graph Hierarchy: Nodes

- 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`:
  - 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

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

- Root **Node** will **always** be a `javafx.scene.layout.Pane` or one of `Pane`'s subclasses
- Different **Pane**s have different built-in layout capabilities to allow easy positioning of UI elements – see below for options!
- 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
  - Scene Graph starts off as a single root Node with no children
  - the root is simply a container, without graphical shape

```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 (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 can we 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
  o by adding/removing Nodes from a Node’s list of children, we can add/remove Nodes from the Scene Graph!
  o later we’ll see how Java supports Lists
Adding UI Elements to the Scene (2/2)

- `getChildren()` returns a **List** of the child **Nodes**
  - 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**!
  - also, `addAll(Nodes... node1, node2, ...)` which takes in **any number of Nodes**
  - allowing **any** number of arguments is a new capability of parameter lists
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);
}

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 with remove(Node node)
  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
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
  
  o **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

/* Within App class */

```java
@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)

- 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 its children as well!
  - `root.getChildren().remove(holder);`
    - makes both **VBox** and its **Label** disappear
Removing a **Node** with children (3/3)

- 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 its children as well!
  - `root.getChildren().remove(holder);`
    - makes both **VBox** and its **Label** disappear
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 would correctly remove `removeLabel` from the VBox `holder`?

A. root.remove(removeLabel);
B. holder.remove(removeLabel);
C. root.getChildren().remove(removeLabel);
D. holder.getChildren().remove(removeLabel);
Outline

• GUIs and JavaFX
• JavaFX Scene Graph Hierarchy
• VBox panes and PaneOrganizers
• Example: ColorChanger
• Event Handling and lambda expressions
• Logical vs. Graphical Containment with JavaFX
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.
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);

width, height
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

```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);
```

//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…
  - remember App class should never have more than a few lines of code!

- Write a PaneOrganizer class where all graphical application logic will live – an example of delegation pattern
  - PaneOrganizer is our new graphical top-level class

- 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
   ```java
   root.getChildren().add(...); or
   root.getChildren().addAll(...);
   ```
Outline

• GUIs and JavaFX
• JavaFX Scene Graph Hierarchy
• VBox panes and PaneOrganizers
• **Example: ColorChanger**
• Event Handling and lambda expressions
• Logical vs. Graphical Containment with JavaFX
Our First JavaFX Application: \textit{ColorChanger}

- Spec: App that contains text reading “CS15 Rocks!” and a Button that randomly changes text’s color with every click

- Useful classes: \texttt{Stage}, \texttt{Scene}, \texttt{VBox}, \texttt{Label}, \texttt{Button}, \texttt{EventHandler}

Process: **ColorChanger**

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

2. Create top-level **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 handler
ColorChanger: App class (1/3)

1. To implement start:

A. Instantiate a PaneOrganizer as top-level class and store it in the local variable organizer

```
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 */
    }
}
```
1. To implement `start:`

A. Instantiate a `PaneOrganizer` as top-level class 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 (3/3)

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`
   
   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 handler
ColorChanger: Our PaneOrganizer Class (1/4)

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() {
        this.root = new VBox();
    }

    public VBox getRoot() {
        return _root;
    }
}
```
**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
   - reminder: this makes root Pane accessible from within App’s start for new Scene(root)

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

    public PaneOrganizer() {
        this.root = new VBox();
    }

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

C. Instantiate Label and Button, passing in String representations of text we want displayed
   - myLabel and btn are local variables because only need to access them from within constructor

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

    public PaneOrganizer() {
        this.root = new VBox();
        Label myLabel = new Label("CS15 Rocks!");
        Button btn = new Button("Random Color");
    }

    public VBox getRoot() {
        return this.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
   - myLabel and btn are local variables because only need to access them from within constructor

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

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

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

    public VBox getRoot() {
        return this.root;
    }
}
```
Containment / Association Structure (1/2)

Scene is always contained in App; but no need to include in your own containment diagrams…
This simplified diagram will suffice!
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 handler.
Generating `javafx.scene.paint.Colors` (1/2)

- Let's first determine what should happen to generate the Label’s random 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

```
0 0 51 102 153 204 255
0 0 51 102 153 204 255
0 0 51 102 153 204 255
255 255 0 153 0 204 255
0 0 255 255 255 255 255
0 255 255 0 255 255 255
```

- `javafx.scene.paint.Color` class 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)`
1. Defining our method to change color of the label:

- `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 myLabel, passing in new random `Color` we’ve created

```java
public void changeLabelColor(Label myLabel) {
    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);
    myLabel.setTextFill(customColor);
}
```
Outline

• GUIs and JavaFX
• JavaFX Scene Graph Hierarchy
• VBox panes and PaneOrganizers
• Example: ColorChanger
• Event Handling and lambda expressions
• Logical vs. Graphical Containment with JavaFX
Responding to User Input

• When should `changeLabelColor` be called?

• 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 (2/3)

- **EventHandler** interface declared as:

  ```
  public interface EventHandler<T extends Event> ...
  ```

  - the code inside literal `< >` is known as a “generic parameter” – this is magic for now
  - lets you **specialize** the interface method declarations to handle one specific 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
EventHandlers (3/3)

- `EventHandler` interface only has one method, the `handle` method
- Parameter of `handle` will match the generic parameter of `EventHandler` type
  - in this case `ActionEvent` since `Buttons` generate `ActionEvents`
  - JavaFX generates the specific event for you and passes it as an argument to your `handle` method
  - for `MouseEvent` and `KeyEvent`, you will need to use the event parameter (during next lecture!)

**Method Summary**

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td><code>handle(T event)</code></td>
</tr>
</tbody>
</table>

Invoked when a specific event of the type for which this handler is registered happens.
Registering an **EventHandler** (1/2)

- How do we let a **Button** know which **EventHandler** to execute when it’s clicked?
- We must **register** the **EventHandler** with the **Button** via the **Button**’s **setOnAction** method so that JavaFX can store the association with the **EventHandler** and call it when the **Button** is clicked
  - note the “generic parameter” `<ActionEvent>` since button clicks generate **ActionEvents**

```java
public final void setOnAction(EventHandler<ActionEvent> value)
Sets the value of the property onAction.

**Property description:**
The button's action, which is invoked whenever the button is fired. This may be due to the user clicking on the button with the mouse, or by a touch event, or by a key press, or if the developer programmatically invokes the **fire()** method.
```
Registering an **EventHandler** (2/2)

```java
public class MyClickHandler implements EventHandler<ActionEvent> {
    private Label myLabel;
    public MyClickHandler(Label myLabel) {
        this.myLabel = myLabel;
    }
    @Override
    public void handle(ActionEvent e) {
        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);
        this.myLabel.setTextFill(customColor);
    }
}
```

1. Write custom **EventHandler** class (**MyClickHandler**), implementing **handle** with previous code to generate **Color**
   - must create an association with the **Label** so the handler knows which **Label** to change

2. In **PaneOrganizer**, register the **EventHandler** with the **Button**, using **setOnAction** method

3. When **Button** is clicked, **handle** method in **MyClickHandler** is passed an event by JavaFX and is then executed
Lambda Expressions (1/3)

• Creating a separate class `MyClickHandler` is not the most efficient solution
  o more complex `EventHandler`s may have tons of associations, all to implement one `handle` method

• Since `EventHandler` interface only has one method, we can use special syntax called a `lambda expression` instead of defining a separate class for implementation of `handle`
Lambda Expressions (2/3)

- **Lambda expression** has different syntax with same semantics as typical method
  - first **parameter list**
  - followed by -
  - then an arbitrarily complex **method body** in curly braces
    - in CS15, lambda expression body will be one line calling another method, typically written yourself in the same class; in this case `changeLabelColor`
    - can omit curly braces when method body is one line

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

    public PaneOrganizer() {
        this.root = new VBox();  
        Label myLabel = new Label("CS15 Rocks!");  
        Button btn = new Button("Random Color");
        this.root.getChildren().addAll(myLabel, btn);
        this.root.setSpacing(8);
        btn.setOnAction((ActionEvent e) ->
            this.changeLabelColor(myLabel));
    }

    public void changeLabelColor(Label myLabel) {
        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);  
        myLabel.setTextFill(customColor);
    }
}
```
Lambda Expressions (3/3)

• Lambda expression shares **scope** with its enclosing method
  - can access **myLabel** or **btn** without setting up a class association

• Lambda expression body is then stored by JavaFX to be called once the button is clicked

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

    public PaneOrganizer() {
        this.root = new VBox();
        Label myLabel = new Label("CS15 Rocks!");
        Button btn = new Button("Random Color");
        this.root.getChildren().addAll(myLabel, btn);
        this.root.setSpacing(8);
        btn.setOnAction((ActionEvent e) ->
            this.changeLabelColor(myLabel));
    }

    public void changeLabelColor(Label myLabel) {
        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);
        myLabel.setTextFill(customColor);
    }
}
```
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.application.Application;

public class App extends Application {
    @Override
    public void start(Stage stage) {
        PaneOrganizer organizer = new PaneOrganizer();
        Scene scene = new Scene(organizer.getRoot(),180,80);
        stage.setScene(scene);
        stage.setTitle("Color Changer");
        stage.show();
    }
}

import javafx.scene.layout.VBox;
import javafx.scene.control.Label;
import javafx.scene.control.Button;
import javafx.event.ActionEvent;
import javafx.scene.paint.Color;

public class PaneOrganizer {
    private VBox root;

    public PaneOrganizer() {
        this.root = new VBox();
        Label myLabel = new Label("CS15 Rocks!");
        Button btn = new Button("Random Color");
        this.root.getChildren().addAll(myLabel,btn);
        this.root.setSpacing(8);
        btn.setOnAction((ActionEvent event) ->
            this.changeLabelColor(myLabel));
    }

    public VBox getRoot() {
        return this.root;
    }

    private void changeLabelColor(Label myLabel) {
        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);
        myLabel.setTextFill(customColor);
    }
}
Outline

• GUIs and JavaFX
• JavaFX Scene Graph Hierarchy
• VBox panes and PaneOrganizers
• Example: ColorChanger
• Event Handling and lambda expressions
• Logical vs. Graphical Containment with JavaFX
• **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

- **Code from today’s lecture** is available on Github – mess around for practice!

- **Fruit Ninja deadlines**
  - Early handin: Sunday 10/10
  - On-time handin: Tuesday 10/12
  - Late handin: Thursday 10/14

- Confused about the Javadocs? Be sure to submit the [Fruit Ninja Javadocs quiz](#) prior to coding to make sure you have a solid grasp on the support code

- We **will** hold TA hours over the long weekend
  - Monday hours will only be conceptual, and may be more limited because they are optional for our TAs

- Debugging hours start today
  - Read the message on Ed for full debugging hours logistics
Topics in Socially Responsible Computing

Cybercrime and Ransomware
Cybercrime

• The use of a computer or online network to commit crimes such as fraud, online image abuse, identity theft, or threats and intimidation
  o Can target Individuals, businesses, education institutes and governments

• Threat to privacy and security
  o Yahoo (2013): 3 billion accounts
  o Alibaba (2019): 1.1 billion pieces of user data
  o LinkedIn (2021): 700 million users
  o Sina Weibo (2020): 538 million accounts
  o Marriott International (2018): 500 million customers

Photo: New Food magazine
Ransomware

• “A form of malware designed to encrypt files on a device, rendering any files and the systems that rely on them unusable” - CISA.gov
  o Ransomware creators then demand ransom in exchange for decryption
• Springhill Medical Center hit by ransomware attack in 2019
  o Computers disabled medical staff cut off from equipment, patient health records inaccessible
  o Teiranni Kidd gave birth to baby born with umbilical cord wrapped around her neck
    ▪ Passed away 9 months later
    ▪ Kidd suing—information about baby’s condition wiped during hack
  o If proven, case would mark first confirmed death from ransomware attack
• In 2017, suspected Russia-backed hackers targeted Baltic energy networks
  o Lithuania, Latvia, Estonia
Ransomware @ Brown

• March 30 2021: suspected cybersecurity attack forced the university to shut down certain computer programs
  o Banner Self Service, Canvas, Workday, Zoom, Google
  o Affected Microsoft Windows-based programs → shut down connections to university’s central data center

• No public information available about privacy leaks affiliated with this event
  o Could have been a terrible bug → happened at multiple universities

Photo: NBC
Future Threats to Cybersecurity

• Speculation that “WWWIII will be fought in cyberspace”
  o “World War III is a guerrilla information war with no division between military and civilian participation” – Marshall McLuhan, Culture is Our Business (1970)

• Three types of cyber warfare attacks
  o Destabilization: attack critical infrastructures: transportation systems, power grids, banking systems, dams, water supplies, and hospitals
  o Sabotage: stop government systems from communicating, enable intelligence theft, threaten national security, contaminate digital systems
  o Data theft: steal personal data of federal employees and civilians
Threats to Cybersecurity

• In 2017, suspected Russia-backed hackers targeted Baltic energy networks
  o Lithuania, Latvia, Estonia
  o Targeted petroleum distribution system

• Israeli government attack Iranian port facility
  o 2020 knocked computers offline
  o Loaded container ships wait along the coast
  o Linked to attack on wastewater plant

• May 2021 Colonial Pipeline Co. attacked
  o Critical pipeline across 5,500 miles of east coast
More reading that may be of interest!

• **A Hospital Hit by Hackers, a Baby in Distress: The Case of the First Alleged Ransomware Death** — WSJ
• **Cyber Warfare - Truth, Tactics And Strategies** — Forbes
• **Cyber Warfare** — rand.org
• **When Cyber War Becomes War** — Forbes
• **Biden administration to convene 30 countries to crack down on ransomware threat** — CNN
• **Cyberattack on Iranian Port Facility** — WSJ
• **Suspected Russia-backed hackers target Baltic energy networks** — Reuters