



Lecture 8

Graphics Part I
Intro to JavaFX

(photo courtesy of Snapchat filters) 1/63

Andreas van Dam © 2019 10/29/19

1

This Lecture

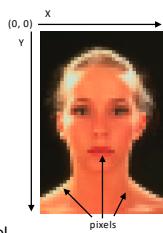
- [GUIs and JavaFX](#)
- [JavaFX Scene Graph Hierarchy](#)
 - Aside: `VBox`
- [Example: ColorChanger](#)
 - Event Handling
 - Private inner classes
 - Random number generation
 - `javafx.scene.paint.Colors`
- [Logical vs. Graphical Containment with JavaFX](#)

2/63

2

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



Former HTA Sam Squires!

3/63

3

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!

4/63

4

Creating Applications from Scratch

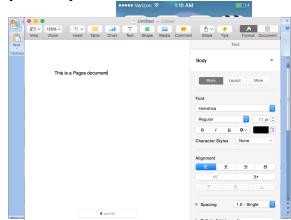
- Until now, TAs took care of graphical components for you
 - our support code defined the relevant classes
- *From now on, you are in charge of this!*
- JavaFX is quite powerful but can be a bit tricky to wrap your head around because of the size of the JavaFX library
 - not to fear, all JavaFX packages, classes, and method descriptions can be found in the [JavaFX guide](#) on our website!

5/63

5

Graphical User Interface (GUIs)

- GUIs provide user- controlled (i.e., graphical) way to send messages to a system of objects, typically your app
- Use JavaFX to create your own GUIs throughout the semester

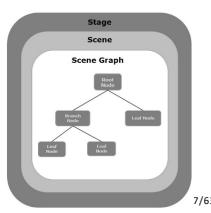


6/63

6

Components of JavaFX application

- Stage
 - **location** (or “window”) where all graphic elements will be displayed
 - Scene
 - **container** for all UI elements to be displayed on a stage
 - scene must be on a stage to be visible
 - Scene Graph
 - family tree of graphical elements
 - Nodes
 - all elements of the Scene Graph
 - graphical representation called a UI element, widget or control (synonyms)



7/63

7

Creating GUIs With JavaFX (1/2)

- **App** class for our main JavaFX application extends the imported **abstract** class `javafx.application.Application`
 - From now on in CS15, we'll begin every project by implementing the **Application** class' **start** method
 - **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** method
 - the **stage** becomes a window for the application when **stage's show** method is called in the **Application** class' **start** method.

```
public class App extends Application {  
    //mainline provided by TAs elided  
    @Override  
    public void start( Stage stage ) {  
        stage.show();  
    }  
}
```



8/63

8

Creating GUIs With JavaFX (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** ?

Process shown in a few slides!

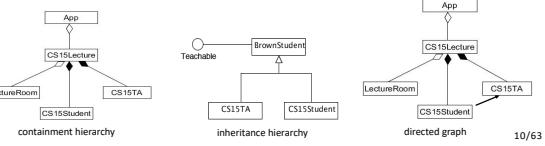
Process shown in

9/63

9

JavaFX Scene Graph Hierarchy (1/3)

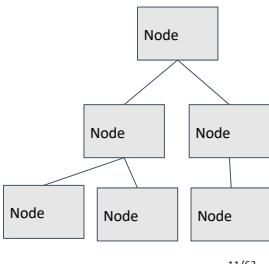
- In JavaFX, contents of the **Scene** (UI elements) are represented as a hierarchical **tree**, known as the Scene Graph
 - you are familiar with some other hierarchies already - **containment** and **inheritance**



10

JavaFX Scene Graph Hierarchy (2/3)

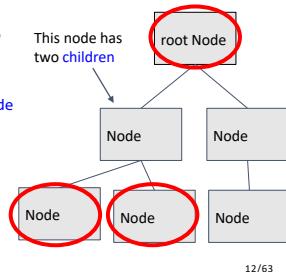
- Think of the Scene Graph as a *family tree of visual elements*
- javafx.scene.Node** is the abstract superclass for all UI elements that can be added to the **Scene**, such as a **Button** or a **Label**
 - all UI elements are concrete subclasses of **Node** (**Button**, **Label**, **Pane**, etc.)
- Each UI component that is added to the Scene Graph as a **Node** gets displayed *graphically*



11

JavaFX Scene Graph Hierarchy (3/3)

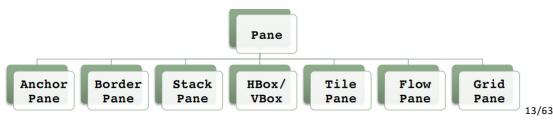
- Each **Node** can have multiple *children* but at most one *parent*
 - child **Nodes** are almost always *graphically contained* in their parent **Node**
 - more on graphical containment later!
- The **Node** at the top of the Scene Graph is called the **root Node**
 - the **root Node** has no parent



12

The root of the Scene

- In CS15, root **Node** will **always** be a `javafx.scene.layout.Pane` or one of `Pane`'s 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



13/63

13

Constructing the Scene Graph (1/3)

- Instantiate root Node

VBox
root

```
public class App extends Application {  
  
    @Override  
    public void start(Stage stage) {  
        //Code to populate Scene Graph  
        VBox root = new VBox();  
  
    }  
}
```

14/63

14

Constructing the Scene Graph (2/3)

- Instantiate root **Node**
 - Then pass it into **Scene** constructor to construct **Scene Graph**

VBox
root

```
graphical shape
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);
    }
}
```

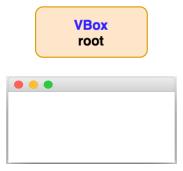
15/63

15

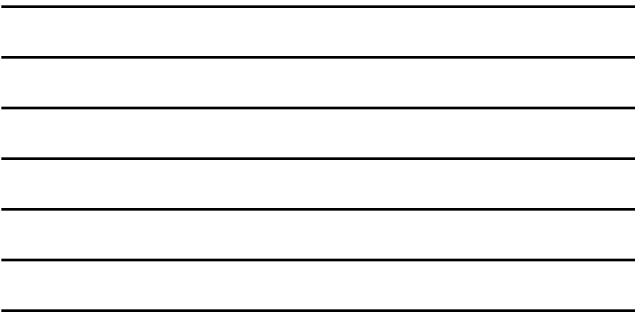
Constructing the Scene Graph (3/3)

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

```
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();  
    }  
}
```



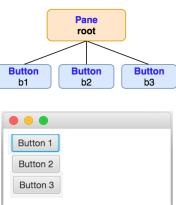
16/63



16

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
 - by adding/removing **Nodes** from a **Node**'s list of children, we can add/remove **Nodes** from the Scene Graph!



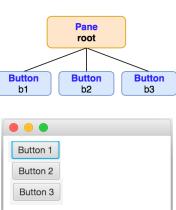
17/63



17

Adding UI Elements to the Scene (2/2)

- `getChildren()` returns a `List` of 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`!
 - can also use `addAll(Nodes... node1, node2, ...)` which takes in *any number of Nodes*
 - 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`



18/63



18

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: 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!

19/63

19

Removing UI Elements from the Scene (2/2)

- Similarly, remove a UI element by removing it from `root`'s children
- 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!

20/63

20

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` – next slide
- This will continue to populate the scene graph!

21/63

21

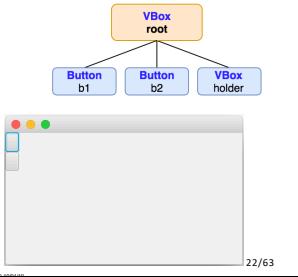
Populating the Scene Graph (2/3)

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

```
/* 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);
}
```



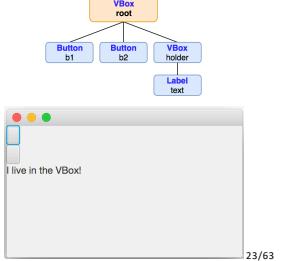
22

Populating the Scene Graph (3/3)

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

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

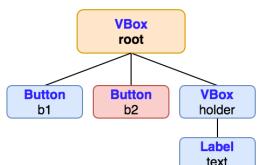
    Button b1 = new Button();
    Button b2 = new Button();
    root.getChildren().addAll(b1,b2);
    VBox holder = new VBox();
    root.getChildren().add(holder);
    Label text = new Label(
        "I live in the VBox!");
    holder.getChildren().add(text);
}
```



23

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**

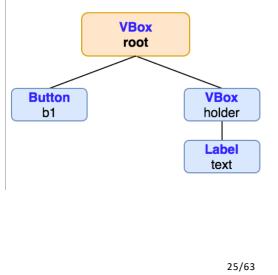


24/63

24

Removing a Node with children (2/3)

- 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!

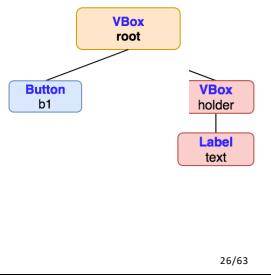


25/63

25

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 `Label` disappear



26/63

26

TopHat Question

```
Given this code:  
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");  
    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);

27/63

27

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
 - the larger the **double** passed in, the more space between the **child** UI elements



28/63

28

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

29/63

29

VBox layout pane (3/5)

- The following code produces the example on the right:

```
VBox root = new VBox();  
  
Button b1 = new Button("Top");  
Button b2 = new Button("Middle");  
Button b3 = new Button("Bottom");
```

```
width, height  
Scene scene = new Scene(root, 200, 200);  
stage.setTitle("Sample VBox");  
stage.setScene(scene);  
stage.show();
```



30/63

30

VBox layout pane (4/5)

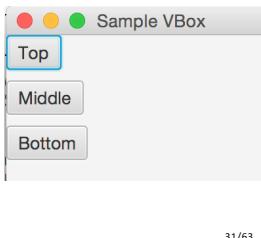
- Adding spacing between children

```
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
```



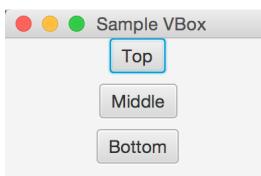
31/63

31

VBox layout pane (5/5)

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

```
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);
```



32/63

32

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...
 - 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` – 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!

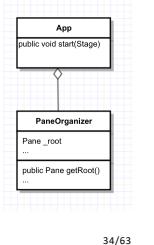
33/63

33

CS15 PaneOrganizer Class (2/2)

Pattern

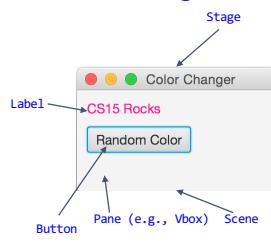
1. App class instantiates a **PaneOrganizer**, which creates root
2. Then App class passes return value from `getRoot()` to **Scene** constructor, so **Scene** has a root
3. Top-level **PaneOrganizer** class instantiates JavaFX UI components (**Button**, **Label**, **Pane**...)
4. These UI components are then added to root **Pane** (and the **Scene**, indirectly) using
`root.getChildren().add(...);` or
`root.getChildren().addAll(...);`



34

Our First JavaFX Application: ColorChanger

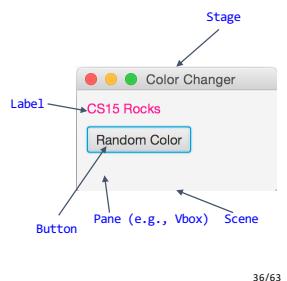
- Spec: App that contains text reading "CS15 Rocks!" and a **Button** that randomly changes text's color with every click
- Useful classes: **Stage**, **Scene**, **VBox**, **Label**, **Button**, **EventHandler**



35

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**



36

ColorChanger: App class (1/3)

- ## 1. To implement `start`:

```
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 */  
    }  
}
```

37/63

37

ColorChanger: App class (2/3)

- ### 1. To implement `start`:

```
public class App extends Application {  
    @Override  
    public void start(Stage stage) {  
        PaneOrganizer organizer = new PaneOrganizer();  
        //Write out PaneOrganizer class later,  
        //where we will instantiate the root Pane /*  
        Scene scene =  
            new Scene(organizer.getRoot(), 80, 80);  
        stage.setScene(scene);  
        stage.show();  
    }  
}
```

38/63

- A. Instantiate a `PaneOrganizer` and store it in the local variable `organizer`
 - B. Instantiate a new `Scene`, passing in:
 - o `root Pane`, accessed through `organizer's` public `getRoot()`
 - o `along with desired width and height of Scene`

38

ColorChanger: App class (3/3)

- ## 1. To implement `start`:

```
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();  
    }  
}
```

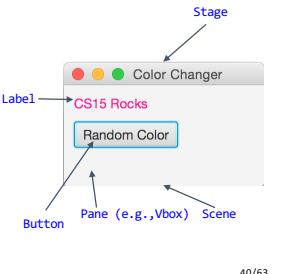
39/63

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

39

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`



40/63

40

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

2. To write `PaneOrganizer` class:

```
public class PaneOrganizer {
    private VBox _root;

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

41/63

41

ColorChanger: Our `PaneOrganizer` Class (2/4)

2. To write `PaneOrganizer` class:

```
public class PaneOrganizer {
    private VBox _root;

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

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

42/63

42

ColorChanger: Our PaneOrganizer Class (3/4)

- ## 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.

```
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;
    }
}
```

43/63

43

ColorChanger: Our PaneOrganizer Class (4/4)

- ## **2. To write `PaneOrganizer` class:**

C. Instantiate `Label` and `Button`, passing in `String` representations of text we

- 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

```
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;
    }
}
```

44/63

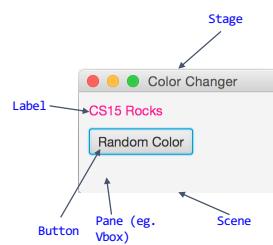
44

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.



45/63

45

Responding to User Input

- Need a way to **respond** to stimulus of **Button** being clicked
 - We refer to this as **Event Handling**
 - a source (**Node**), such as a **Button**, generates an **Event** (such as a mouse click) and notifies all registered **EventHandler** objects
 - **EventHandler** is an interface, so all classes that implement **EventHandler** must implement **handle(Event event)** method, which defines response to event
 - note that **handle(Event event)** is called by JavaFX, not the programmer



46/63

46

EventHandlers (1/3)

- Button click causes JavaFX to generate a `javafx.event.ActionEvent`
 - `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`
 - 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 literal `< >` is known as a "generic parameter" – this is magic for now
 - 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 (next slide)

47/63

47

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
 - literally, < > included!!

48/63

48

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`

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

    @Override
    public void handle(ActionEvent e) {
        // code to change _myLabel Label
    }
}

//elsewhere in program
public class PaneOrganizer {
    //instance variables 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
```

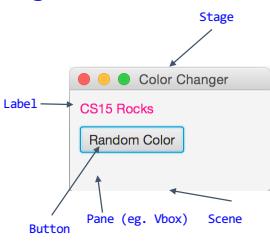
49/63

- To tell this new `ClickHandler` to listen for the `Button`'s `ActionEvent`, register `Button` with the `ClickHandler` by calling `btn.setOnAction(handler)`, passing in an instance of our `ClickHandler` class
 - the mechanics of handing off the event to the handler happen under hood of JavaFX

49

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`



50/63

- ColorChanger: ClickHandler class

3. Defining our custom EventHandler, ClickHandler:

```
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  
    }  
}
```

ClickHandler must listen for click event and respond to it by changing the color of "CS15 Rocks!" Label

How will ClickHandler access Label?

 - o multiple ways to do this:
 - could have ClickHandler constructor take in a Label as a parameter
 - o this works, but is there a better way?

51/63

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
 - Introducing private inner classes!
 - 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
 - **private inner classes have access to instance variables/methods of the class that contains them (that declared them)**
 - inner classes are a convenient and safe shortcut - don't require a file



52/63

52

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`

53/63

53

ColorChanger: ClickHandler Private Inner Class (1/2)

- ```
public class PaneOrganizer {
 private VBox _root;
 private Label _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

 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> {
 @Override
 public void handle(ActionEvent event) {
 _label.setText("Clicked!");
 }
 }
}
```

54/63

54

### ColorChanger: ClickHandler Private Inner Class (2/2)

```

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

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

 Now must
 implement handle
 method
 public VBox getRoot() {
 return _root;
 }

 How will
 ClickHandler
 generate a random
 color whenever
 btn's ActionEvent
 is detected?
 private class ClickHandler implements EventHandler<ActionEvent> {
 public ClickHandler() { //code and annotations elided
 public void handle(ActionEvent event) {
 //implementation elided for now
 }
 }
 }
}

```

55/63

55

### Generating javafx.scene.paint.Colors

- 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
- `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)`

56

### ColorChanger: Our EventHandler, ClickHandler

```

3. Defining our custom
EventHandler, ClickHandler:
 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);
 }
 }
}

```

56/63

57

### 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 S49)

```
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
 }
}
```

58/63

58

### The Whole App: ColorChanger

```
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());
 }

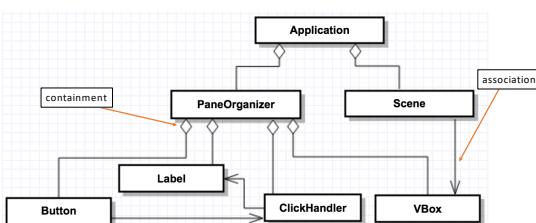
 public VBox getRoot() {
 return _root;
 }

 private class ClickHandler
 implements EventHandler<ActionEvent> {
 @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);
 }
 }
}
```

59/63

59

### Putting It All Together



60/63

60

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

61/63

61

**Announcements**

- FruitNinja deadlines:
  - Early: Friday, 10/4 at 11:59pm
  - On-time: Sunday, 10/6 at 11:59pm
  - Late: Tuesday, 10/8 at 11:59pm
- Sections will be a Design Discussion this week!
- 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 material will be covered by Graphics III

62/63

62