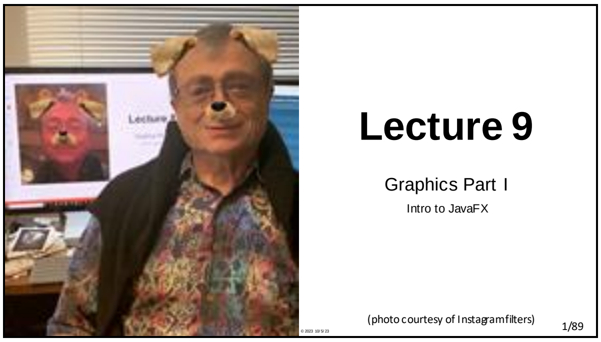
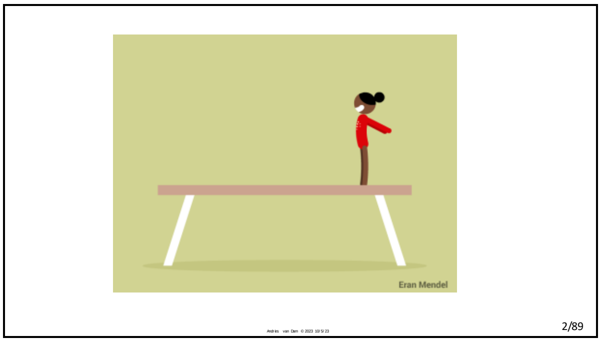


Lecture 9

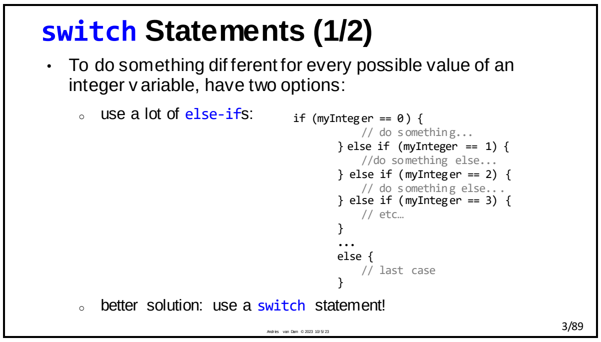
Graphics Part I

Intro to JavaFX

(photo courtesy of Instagramfilters)



Eran Mendel

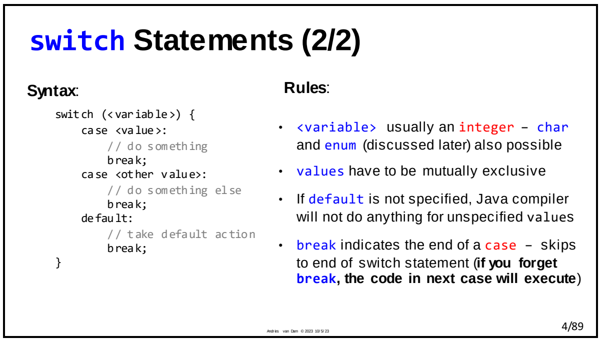


switch Statements (1/2)

- To do something different for every possible value of an integer variable, have two options:
  - use a lot of else-ifs:

```
if (myInteger == 0) {
        // do something...
    } else if (myInteger == 1) {
        //do something else...
    } else if (myInteger == 2) {
        // do something else...
    } else if (myInteger == 3) {
        // etc…
    }
    ...
    else {
        // last case
    }
```

  - better solution: use a switch statement!



switch Statements (2/2)

Syntax:

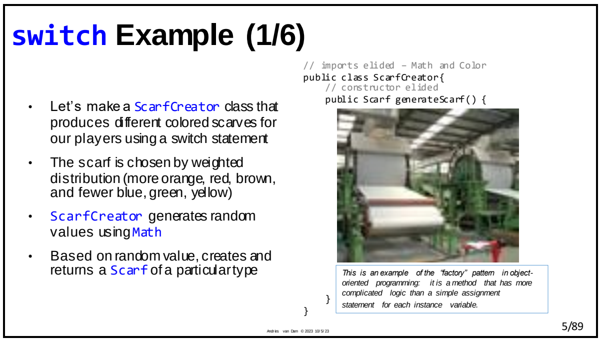
```
switch (<variable>) {
    case <value>:
        // do something
        break;
    case <other value>:
        // do something else
        break;
    default:
        // take default action
        break;
}
```

Rules:

- `<variable>` usually an `integer` – `char` and `enum` (discussed later) also possible
- `values` have to be mutually exclusive
- If `default` is not specified, Java compiler will not do anything for unspecified `values`
- `break` indicates the end of a `case` – skips to end of switch statement (**if you forget `break`, the code in next case will execute**)

Andries van Dam © 2023 10/5/23

4/89



switch Example (1/6)

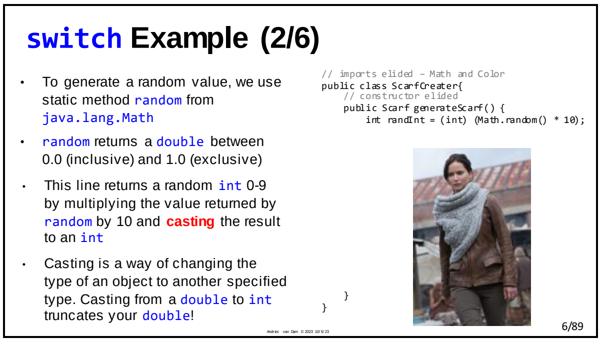
- Let's make a `ScarfCreator` class that produces different colored scarves for our players using a switch statement
- The scarf is chosen by weighted distribution (more orange, red, brown, and fewer blue, green, yellow)
- `ScarfCreator` generates random values using `Math`
- Based on random value, creates and returns a `Scarf` of a particular type

```
// imports elided - Math and Color
public class ScarfCreator{
    // constructor elided
    public Scarf generateScarf() {

    }
}
```

This is an example of the "factory" pattern in object-oriented programming: it is a method that has more complicated logic than a simple assignment statement for each instance variable.

Andries van Dam © 2023 10/5/23

5/89



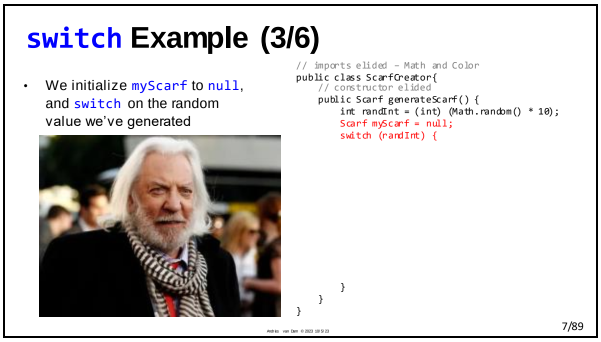
switch Example (2/6)

- To generate a random value, we use static method `random` from `java.lang.Math`
- `random` returns a `double` between 0.0 (inclusive) and 1.0 (exclusive)
- This line returns a random `int` 0-9 by multiplying the value returned by `random` by 10 and **casting** the result to an `int`
- Casting is a way of changing the type of an object to another specified type. Casting from a `double` to `int` truncates your `double`!

```
// imports elided - Math and Color
public class ScarfCreater{
    // constructor elided
    public Scarf generateScarf() {
        int randInt = (int) (Math.random() * 10);

    }
}
```

Andries van Dam © 2023 10/5/23

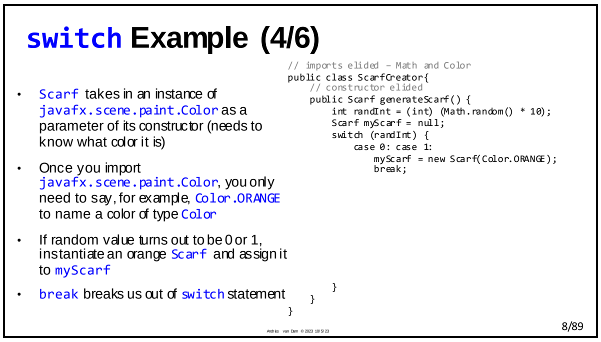
6/89



switch Example (3/6)

- We initialize myScarf to null, and switch on the random value we've generated

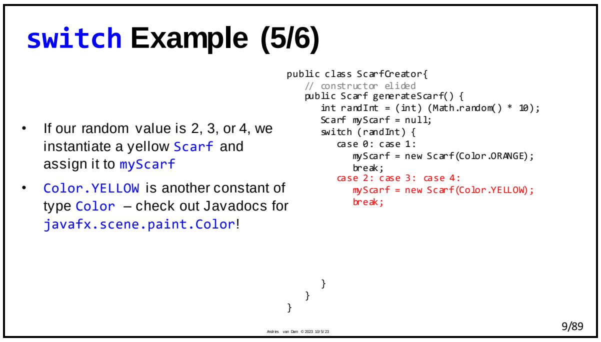
```
// imports elided - Math and Color
public class ScarfCreator{
    // constructor elided
    public Scarf generateScarf() {
        int randInt = (int) (Math.random() * 10);
        Scarf myScarf = null;
        switch (randInt) {











        }
    }
}
```



switch Example (4/6)

- Scarf takes in an instance of javafx.scene.paint.Color as a parameter of its constructor (needs to know what color it is)
- Once you import javafx.scene.paint.Color, you only need to say, for example, Color.ORANGE to name a color of type Color
- If random value turns out to be 0 or 1, instantiate an orange Scarf and assign it to myScarf
- break breaks us out of switch statement

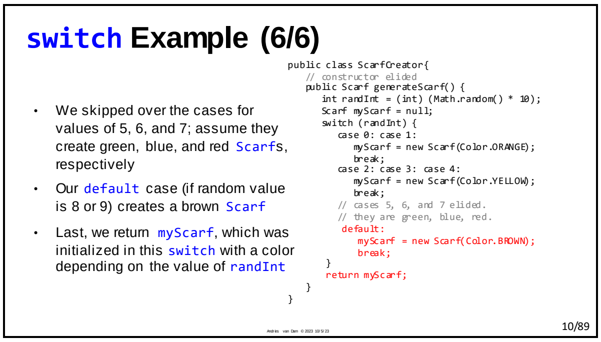
```
// imports elided - Math and Color
public class ScarfCreator{
    // constructor elided
    public Scarf generateScarf() {
        int randInt = (int) (Math.random() * 10);
        Scarf myScarf = null;
        switch (randInt) {
            case 0: case 1:
                myScarf = new Scarf(Color.ORANGE);
                break;









        }
    }
}
```



switch Example (5/6)

- If our random value is 2, 3, or 4, we instantiate a yellow Scarf and assign it to myScarf
- Color.YELLOW is another constant of type Color – check out Javadocs for javafx.scene.paint.Color!

```
public class ScarfCreator{
   // constructor elided
   public Scarf generateScarf() {
      int randInt = (int) (Math.random() * 10);
      Scarf myScarf = null;
      switch (randInt) {
         case 0: case 1:
            myScarf = new Scarf(Color.ORANGE);
            break;
         case 2: case 3: case 4:
            myScarf = new Scarf(Color.YELLOW);
            break;






      }
   }
}
```



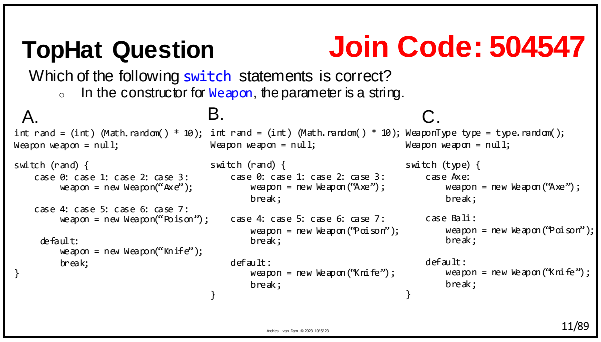
switch Example (6/6)

- We skipped over the cases for values of 5, 6, and 7; assume they create green, blue, and red Scarfs, respectively
- Our default case (if random value is 8 or 9) creates a brown Scarf
- Last, we return myScarf, which was initialized in this switch with a color depending on the value of randInt

```
public class ScarfCreator{
    // constructor elided
    public Scarf generateScarf() {
        int randInt = (int) (Math.random() * 10);
        Scarf myScarf = null;
        switch (randInt) {
            case 0: case 1:
                myScarf = new Scarf(Color.ORANGE);
                break;
            case 2: case 3: case 4:
                myScarf = new Scarf(Color.YELLOW);
                break;
            // cases 5, 6, and 7 elided.
            // they are green, blue, red.
             default:
                myScarf = new Scarf(Color.BROWN);
                break;
        }
        return myScarf;
    }
}
```

Andries van Dam © 2023 10/5/23

10/89



TopHat Question

Join Code: 504547

Which of the following switch statements is correct?

- In the constructor for Weapon, the parameter is a string.

A.

```
int rand = (int) (Math.random() * 10);
Weapon weapon = null;

switch (rand) {
    case 0: case 1: case 2: case 3:
        weapon = new Weapon("Axe");

    case 4: case 5: case 6: case 7:
        weapon = new Weapon("Poison");

     default:
        weapon = new Weapon("Knife");
        break;
}
```

B.

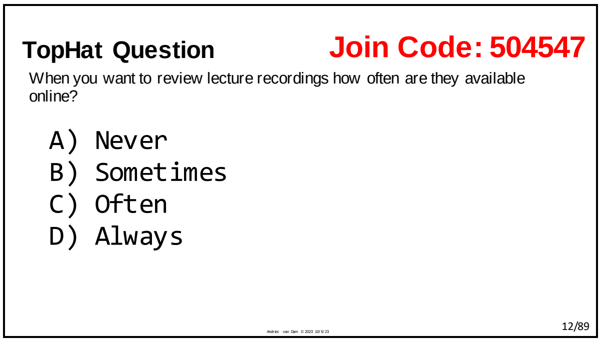
```
int rand = (int) (Math.random() * 10);
Weapon weapon = null;

switch (rand) {
    case 0: case 1: case 2: case 3:
        weapon = new Weapon("Axe");
        break;

    case 4: case 5: case 6: case 7:
        weapon = new Weapon("Poison");
        break;

    default:
        weapon = new Weapon("Knife");
        break;
}
```

C.

```
WeaponType type = type.random();
Weapon weapon = null;

switch (type) {
    case Axe:
        weapon = new Weapon("Axe");
        break;

    case Bali:
        weapon = new Weapon("Poison");
        break;

    default:
        weapon = new Weapon("Knife");
        break;
}
```

Andries van Dam © 2023 10/5/23

11/89



TopHat Question

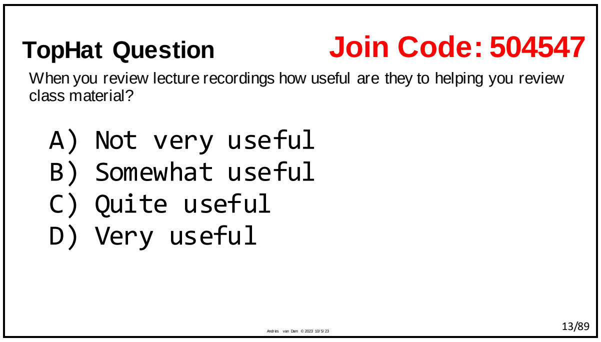
Join Code: 504547

When you want to review lecture recordings how often are they available online?

A) Never
B) Sometimes
C) Often
D) Always

Andries van Dam © 2023 10/5/23

12/89

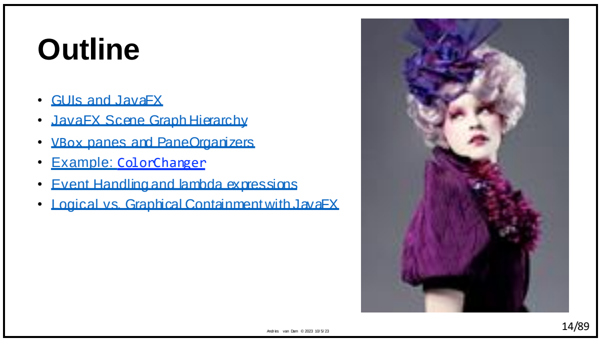


TopHat Question

Join Code: 504547

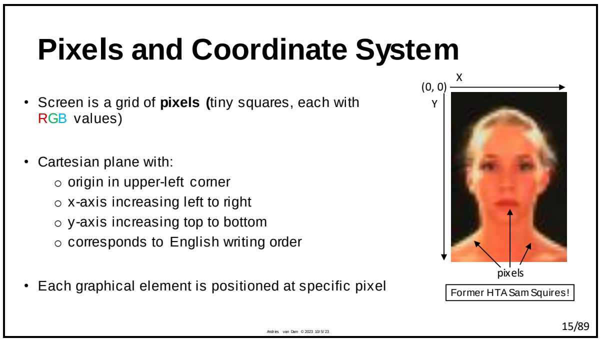
When you review lecture recordings how useful are they to helping you review class material?

A) Not very useful
B) Somewhat useful
C) Quite useful
D) Very useful



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 values)
- 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!

What is JavaFX?

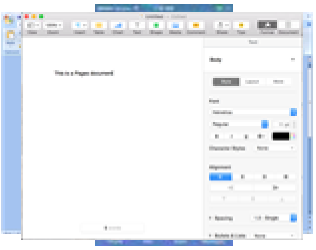
- Usually don’t want to program at the pixel level – far too tedious!
- JavaFX is a set of graphics and media packages enabling developers to design, create, and test powerful graphical applications for desktop, web, and mobile devices
- JavaFX is an API (Application Programming Interface) to a graphics 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
  - 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!

Graphical User Interface (GUIs)

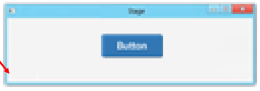
- GUIs provide user-controlled (i.e., graphical) way to send messages to a system of instances, typically your app
- Use JavaFX to create your own GUIs throughout the semester



This is a Pages document

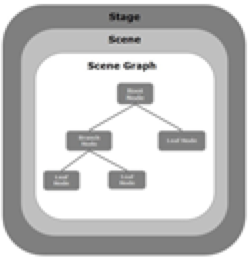
Components of JavaFX application (1/2)

- Stage
  - location (or “window”) where all graphic elements will be displayed
  - blue border with “Stage” label and minimize, maximize and close icons – the “decoration”
- Scene
  - scene (grey interior portion) *must be* on a stage to be visible
  - container for all UI (User Interface) elements to be displayed on a stage
  - UI elements include Panes, Labels, Shapes, etc., like the Button shown



Components of JavaFX application (2/2)

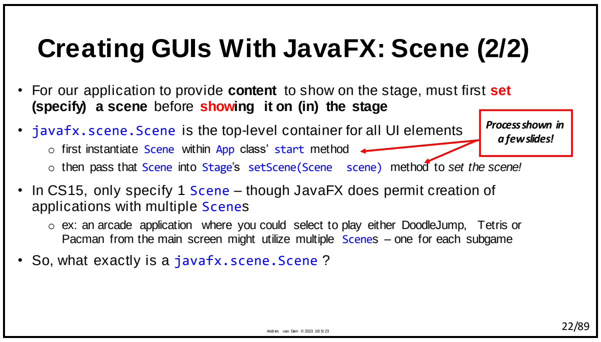
- Scene Graph
  - family tree of graphical elements
- Nodes
  - all elements of the Scene Graph
  - can have multiple children or none
  - graphical representation called a UI element, widget, or control (synonyms)



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 imported javafx.stage.Stage class, which is passed into start()
  - when start() calls stage’s show(), stage becomes a window for the application
- All this automagic reminds us of Main

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

}
```



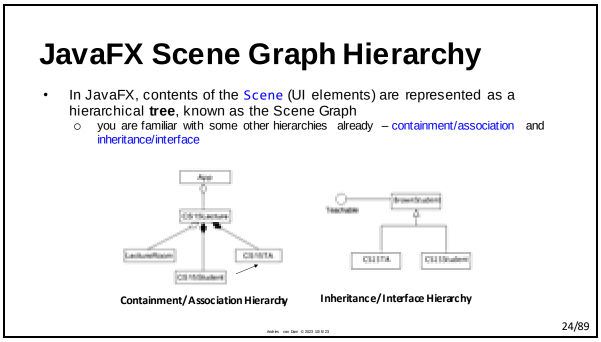
Creating GUIs With JavaFX: Scene (2/2)

- For our application to provide **content** 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 `Scene`s
  - ex: an arcade application where you could select to play either DoodleJump, Tetris or Pacman from the main screen might utilize multiple `Scene`s – one for each subgame
- So, what exactly is a `javafx.scene.Scene` ?

Process shown in a few slides!

Outline

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

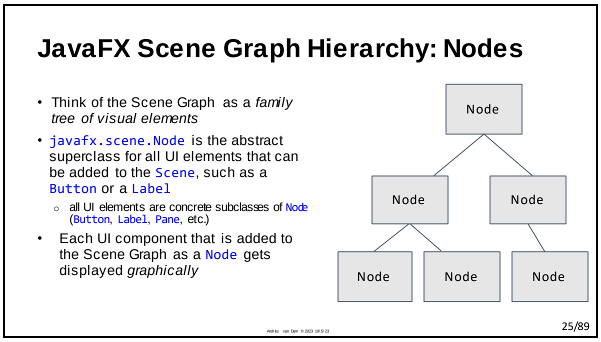


JavaFX Scene Graph Hierarchy

- 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

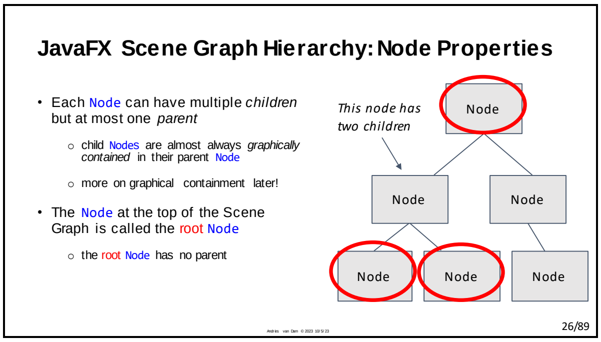
Containment/Association Hierarchy

Inheritance/Interface Hierarchy



JavaFX Scene Graph Hierarchy: Nodes

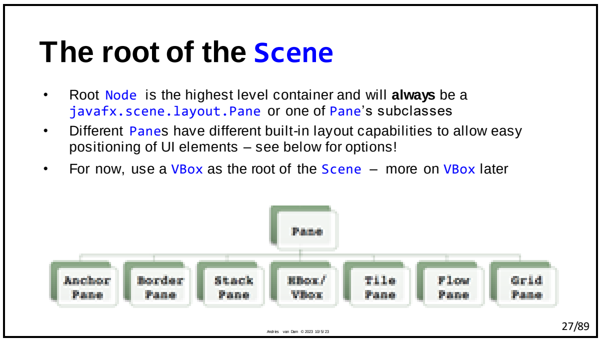
- Think of the Scene Graph as a *family tree of visual elements*
- javafx.scene.Node is the abstract superclass for all UI elements that can be added to the Scene, such as a Button or a Label
  - all UI elements are concrete subclasses of Node (Button, Label, Pane, etc.)
- Each UI component that is added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



JavaFX Scene Graph Hierarchy: Node Properties

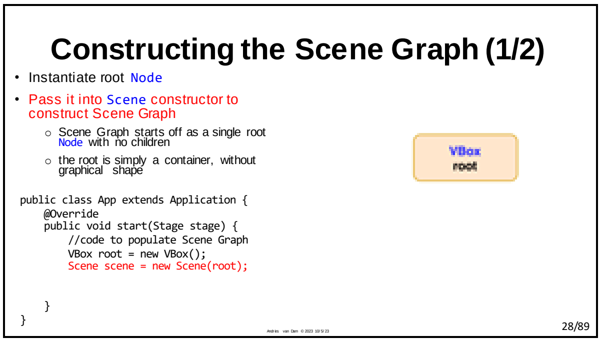
- Each Node can have multiple *children* but at most one *parent*
  - child Nodes are almost always *graphically contained* in their parent Node
  - more on graphical containment later!
- The Node at the top of the Scene Graph is called the root Node
  - the root Node has no parent

This node has two children



The root of the Scene

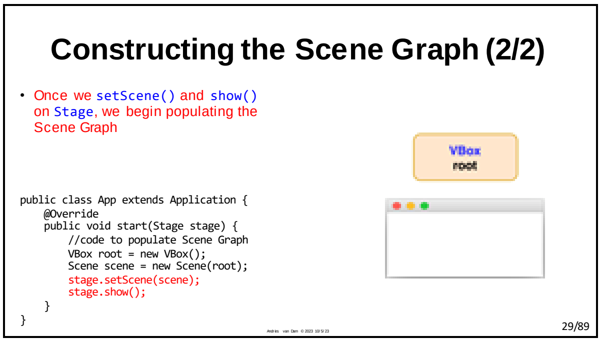
- Root Node is the highest level container and will **always** be a javafx.scene.layout.Pane or one of Pane's subclasses
- Different Panes have different built-in layout capabilities to allow easy positioning of UI elements – see below for options!
- For now, use a VBox as the root of the Scene – more on VBox later



Constructing the Scene Graph (1/2)

- 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

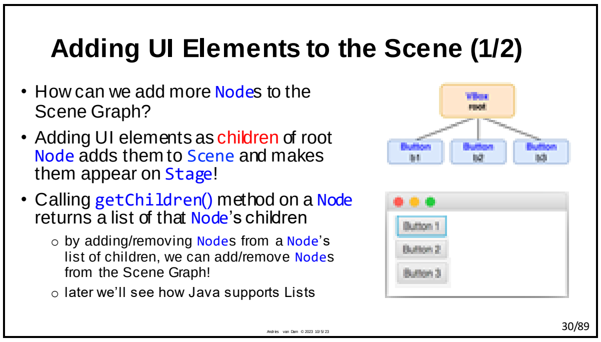
```
public class App extends Application {
    @Override
    public void start(Stage stage) {
        //code to populate Scene Graph
        VBox root = new VBox();
        Scene scene = new Scene(root);


    }
}
```



Constructing the Scene Graph (2/2)

- 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();
    }
}
```

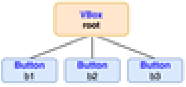


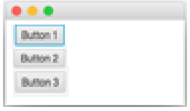
Adding UI Elements to the Scene (1/2)

- 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
  - 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

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



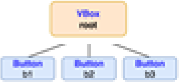


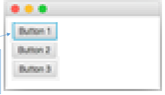
root.getChildren().add(…) in action

- 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);
    }
```

Order matters- order buttons added effects order displayed (b1, b2, b3) vs. (b2, b1, b3)



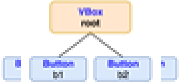


Note the default button selection in blue

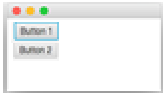
Remember double dot method call shorthand?
root.getChildren() returns a List of root's children. Rather than storing that returned List in a variable and calling add(…) on that variable, we simplify code by calling add(…) directly on returned List of children!

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)
  - note: order of children doesn't matter when removing elements since you specify their variable names
- Let's remove third Button*



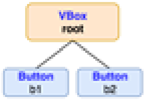
```
/* Within App class */
    @Override
    public void start(Stage stage) {
        //code for setting root, stage, scene elided

        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);
    }
```

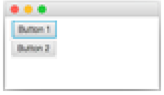


*Note: not a typical design choice to add and then remove a Node in the same code block!

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!

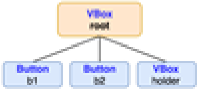




Populating the Scene Graph (2/3)

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

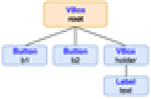
```
/* Within App class */
    @Override
    public void start(Stage stage) {
        //code for setting scene elided

        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);
    }
```

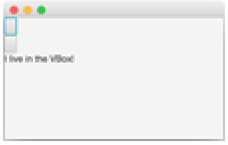


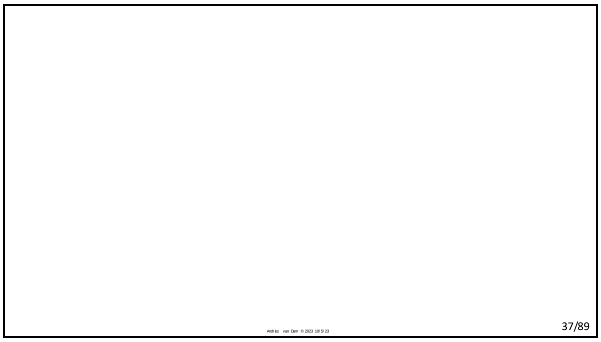
Populating the Scene Graph (3/3)

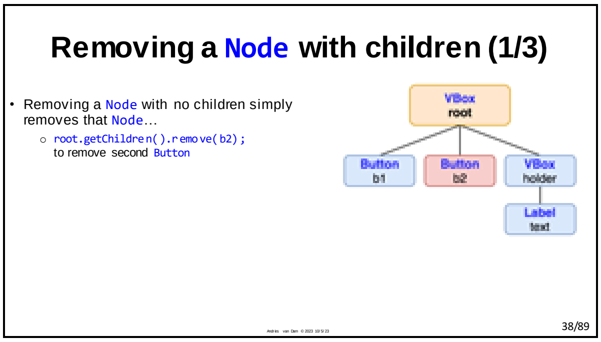
- Next, add Label to Scene as child of new VBox

```
/* 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);
      }
```





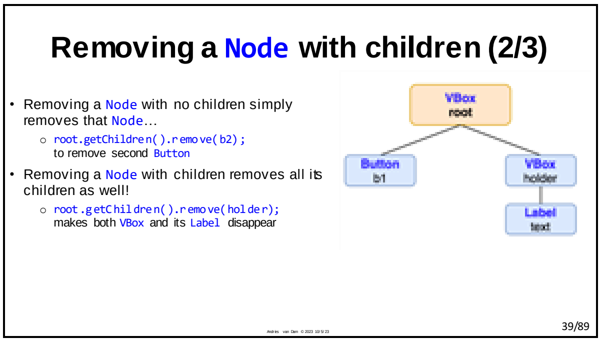




Removing a Node with children (1/3)

- Removing a Node with no children simply removes that Node…
  - ○ `root.getChildren().remove(b2);` to remove second Button

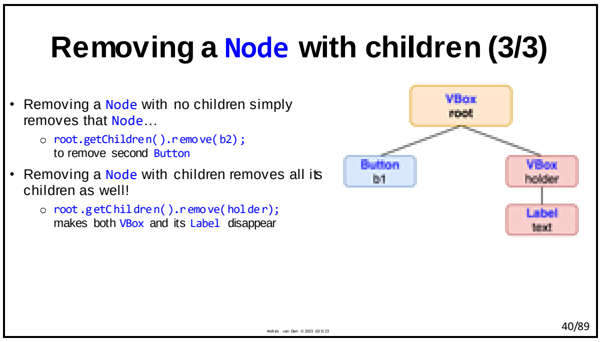
VBox
root
Button
b1
Button
b2
VBox
holder
Label
text



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

VBox
root
Button
b1
VBox
holder
Label
text

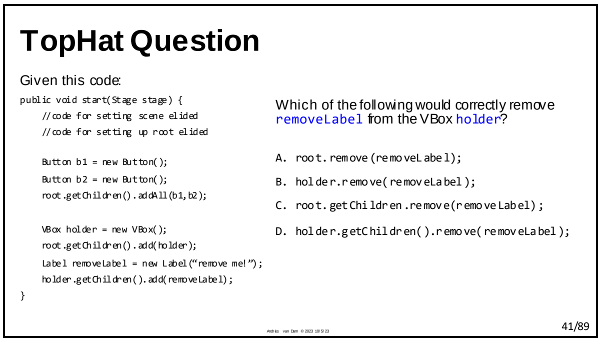


Removing a Node with children (3/3)

- 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

Andries van Dam © 2023 10/5/23

40/89



TopHat Question

Given this code:

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

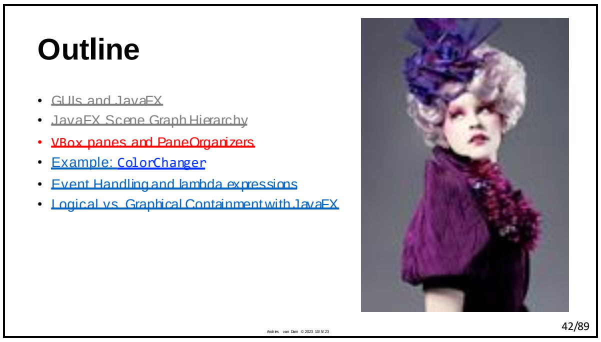
B. holder.remove(removeLabel);

C. root.getChildren.remove(removeLabel);

D. holder.getChildren().remove(removeLabel);

Andries van Dam © 2023 10/5/23

41/89



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

Andries van Dam © 2023 10/5/23

42/89

VBox layout pane (1/5)

- So what exactly is a VBox?
- VBox **is a** Pane that 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

VBox layout pane (2/5)

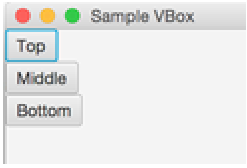
- Can also set positioning of entire vertical column of children
- Default positioning for the vertical column is in TOP_LEFT of VBox (Top Vertically, Left Horizontally)
  - can change Vertical/Horizontal positioning of column using VBox's setAlignment(Pos position) method, passing in a javafx.geometry.Pos constant – javafx.geometry.Pos is a class of enums (more on these later!), 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:

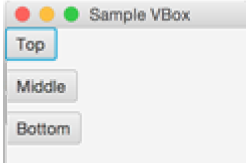
```
VBox root = new VBox();

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, 300, 200);
stage.setTitle("Sample VBox");
stage.setScene(scene);
stage.show();
```



VBox layout pane (4/5)

- Adding spacing between children

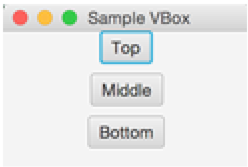
```
VBox root = new VBox();

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

root.setSpacing(8);

//code for setting the Scene elided
```



VBox layout pane (5/5)

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

```
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
```



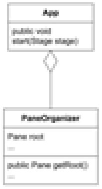
CS15 PaneOrganizer Class (1/2)

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

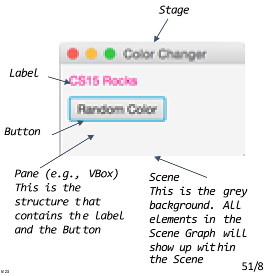


Outline

- 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: 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

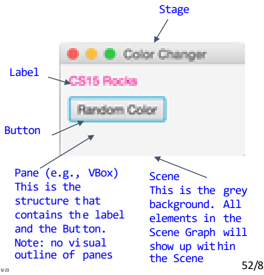


Pane (e.g., VBox)
This is the structure that contains the Label and the Button

Scene
This is the grey background. All elements in the Scene Graph will show up within the Scene

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



Pane (e.g., VBox)
This is the structure that contains the label and the Button.
Note: no visual outline of panes

Scene
This is the grey background. All elements in the Scene Graph will show up within the Scene

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 */




        }

}
```

ColorChanger: App class (2/3)

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

```
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);

    }


}
```

root width height

ColorChanger: App class (3/3)

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:

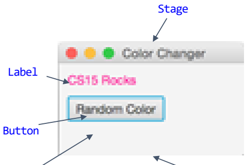
- root Pane, accessed through organizer's public getRoot()
- along with desired width and height of Scene

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

```
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) – 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



Pane (e.g., VBox) This is the structure that contains the label and the Button

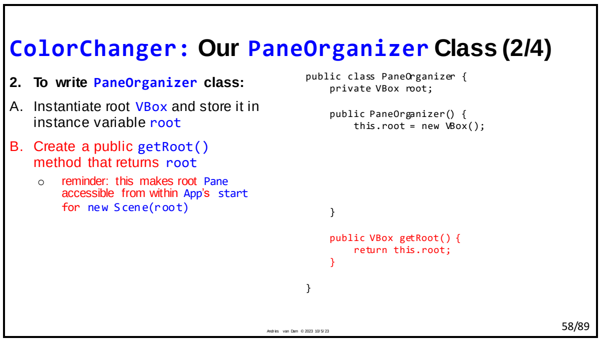
Scene This is the grey background. All elements in the Scene Graph will show up within the Scene

ColorChanger: Our PaneOrganizer Class (1/4)

2. **To write PaneOrganizer class:**

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

```
public class PaneOrganizer {
    private VBox root;

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



    }



}
```



ColorChanger: Our PaneOrganizer Class (2/4)

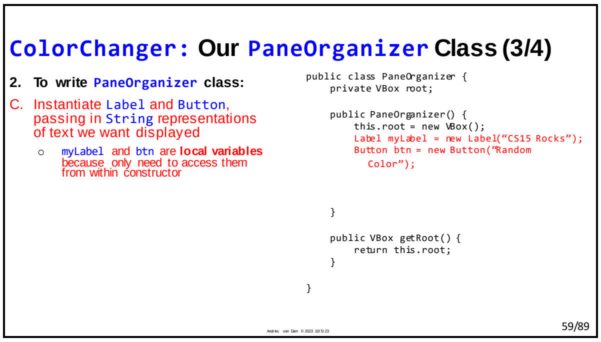
2. **To write PaneOrganizer class:**

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

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

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

```
public class PaneOrganizer {
    private VBox root;

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



    }

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

}
```



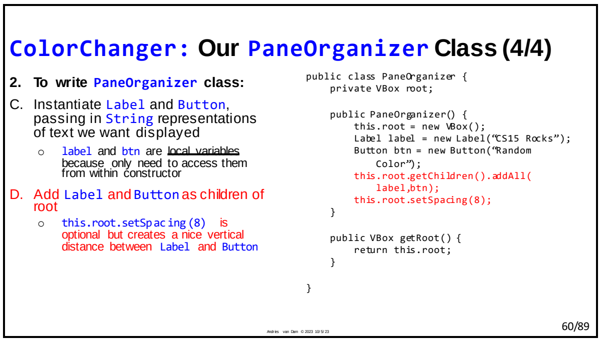
ColorChanger: Our PaneOrganizer Class (3/4)

2. **To write PaneOrganizer class:**

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

- myLabel and btn are **local variables** because only need to access them from within constructor

```
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

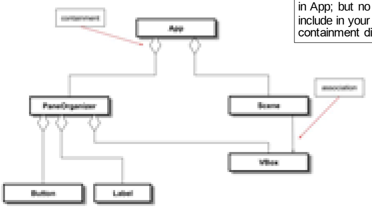
- label and btn are <u>local variables</u> because only need to access them from within constructor

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

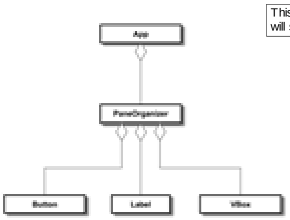
```
public class PaneOrganizer {
    private VBox root;

    public PaneOrganizer() {
        this.root = new VBox();
        Label label = new Label(“CS15 Rocks”);
        Button btn = new Button(“Random
            Color”);
        this.root.getChildren().addAll(
            label,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…

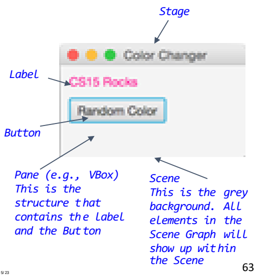
Containment / Association Structure (2/2)



This simplified diagram will suffice!

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

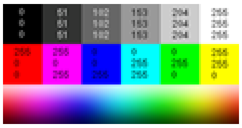


Pane (e.g., VBox)
This is the structure that contains the label and the Button

Scene
This is the grey background. All elements in the Scene Graph will show up within the Scene

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



- 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)

Generating javafx.scene.paint.Colors (2/2)

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

```
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 (like stimulus-response behavioral learning theory in psychology)
- 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 instances of `EventHandler`
  - `EventHandler` is an interface, so all classes that implement `EventHandler` must implement its `handle(Event event)` method, which defines response to event
  - note that `handle(Event event)` is called by JavaFX, not the programmer

EventHandlers (1/3)

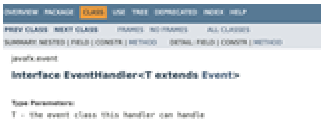
- `Button` click causes JavaFX to generate a `javafx.event.ActionEvent`
  - `ActionEvent` is only one of many JavaFX `EventType`s that are subclasses of `Event` class
- Classes that implement `EventHandler` interface can polymorphically handle any subclass of `Event`
  - when a class implements `EventHandler` interface, it must specify what type of `Event` it should know how to handle
  - how do we do this?

EventHandlers (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



javafx.event

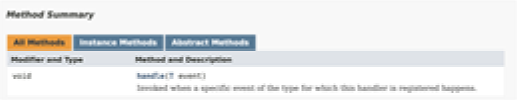
Interface EventHandler<T extends Event>

Type Parameters:

T - the event class this handler can handle

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
  - Note we don't actually use the data contained in an ActionEvent parameter for button click handlers, but for MouseEvents and KeyEvents, you will need to use the event parameter (during next lecture!)



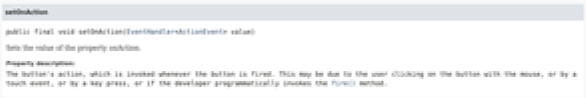
Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method and Description
void	handle(T event) 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



setOnAction

```
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)

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 ActionEvent by JavaFX and is then executed

```
public class MyClickHandler implements EventHandler<ActionEvent> {
    private Label label;
    public MyClickHandler(Label myLabel) {
        this.label = 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.label.setTextFill(customColor);
    }
}

public class PaneOrganizer {

    public PaneOrganizer() {
        // previous code elided
        Label label = new Label("CS15 Rocks");
        Button btn = new Button("Random Color");
        btn.setOnAction(new MyClickHandler(label));
    }
}
```

Lambda Expressions (1/3)

- Creating a separate class `MyClickHandler` is not the most efficient solution
  - more complex `EventHandler`s may have tons of associations with other nodes, 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

```
public class PaneOrganizer {
    private VBox root;

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

    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);
    }
}
```

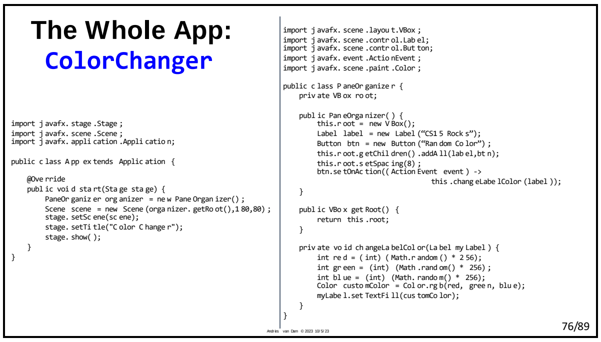
parameter

method body

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

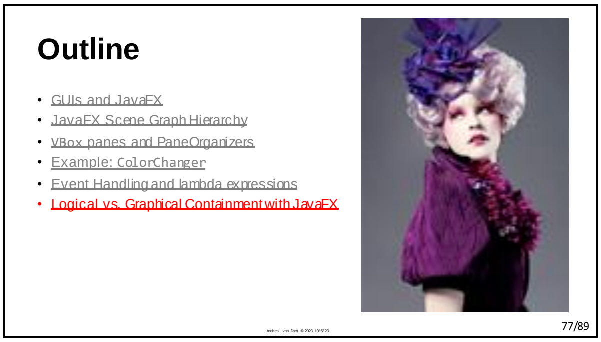
```
public class PaneOrganizer {
    private VBox root;

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

    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);
    }
}
```



The Whole App: ColorChanger

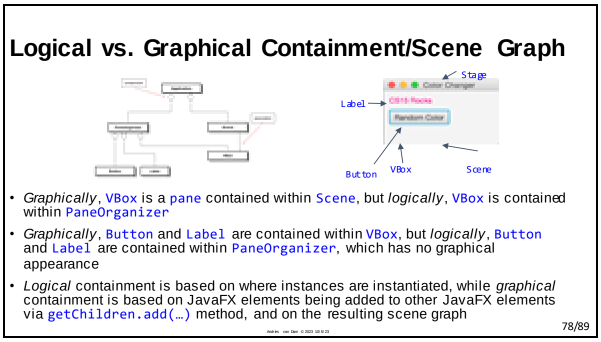
```
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.application.Application;

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

    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);
    }
}
```



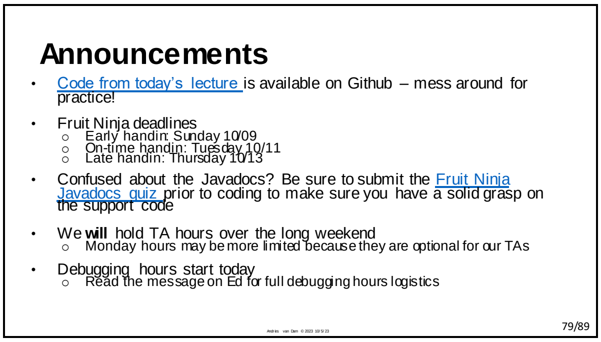
Outline

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



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



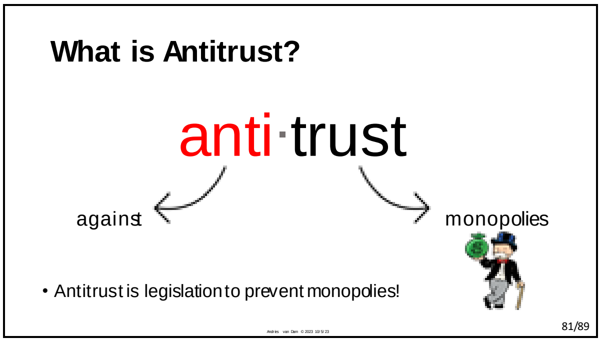
Announcements

- Code from today's lecture is available on Github – mess around for practice!
- Fruit Ninja deadlines
  - Early handin: Sunday 10/09
  - On-time handin: Tuesday 10/11
  - Late handin: Thursday 10/13
- 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 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 SRC: Antitrust and Regulating Big Tech

CS15 Fall 2023

ETHICS IN BIG TECH

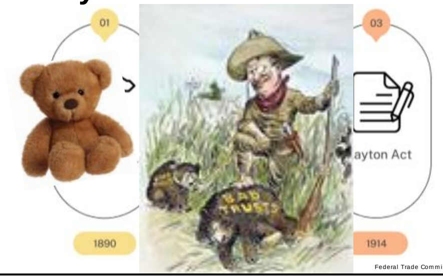


What is Antitrust?

anti·trust
against
monopolies

- Antitrust is legislation to prevent monopolies!

History of US Antitrust



Federal Trade Commission, Clifford Berryman, 190

Traditional antitrust policy needs to evolve

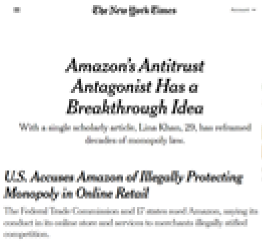
Some platforms are more popular than others

Platform use evolves quickly and often unpredictably

Price-based regulation doesn't work on free platforms

Image source: Freepik, X.

Lina Khan (current chair of the FTC)



The New York Times

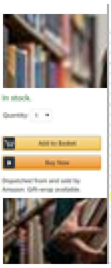
Amazon's Antitrust Antagonist Has a Breakthrough Idea

With a single scholarly article, Lina Khan, 29, has reframed decades of monopoly law.

U.S. Accuses Amazon of Illegally Protecting Monopoly in Online Retail

The Federal Trade Commission and 17 states sued Amazon, saying its conduct in its online store and services to merchants illegally stifled competition.

NYTimes, September 26, 2023



In stock.

Quantity: 1

Add to Basket

Buy Now

Dispatched from and sold by Amazon. Gift-wrap available.

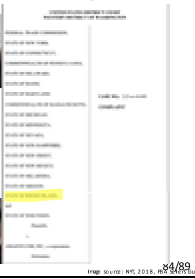
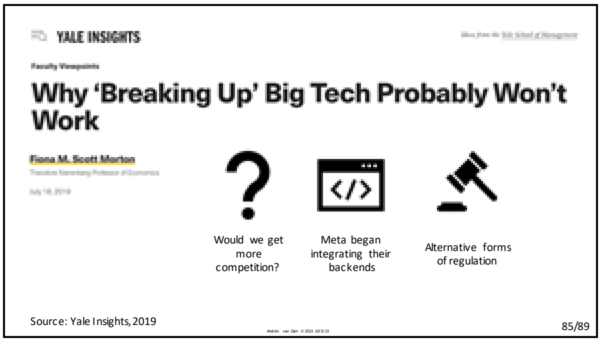


Image source: NYT, 2018, FBA Sellers Gui



YALE INSIGHTS

Why 'Breaking Up' Big Tech Probably Won't Work

Fiona M. Scott Morton

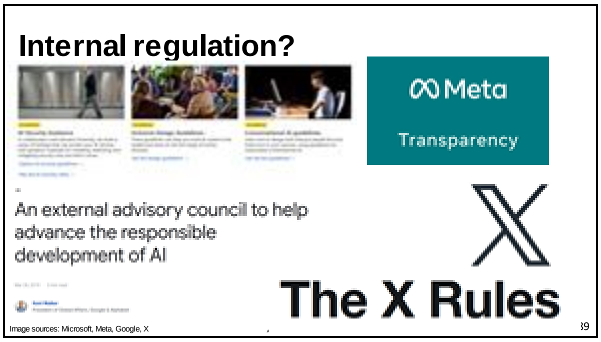
Would we get more competition?

Meta began integrating their backends

Alternative forms of regulation

Source: Yale Insights, 2019

85/89

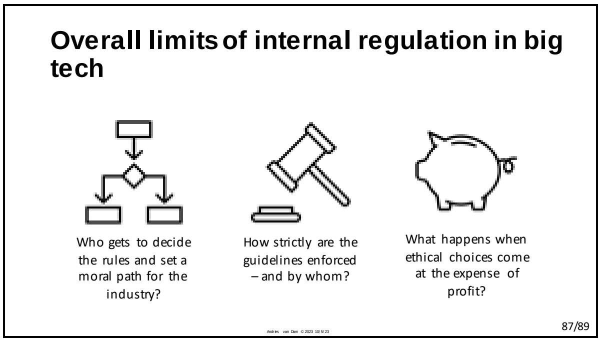


Internal regulation?

An external advisory council to help advance the responsible development of AI

The X Rules

Image sources: Microsoft, Meta, Google, X



Overall limits of internal regulation in big tech

Who gets to decide the rules and set a moral path for the industry?

How strictly are the guidelines enforced – and by whom?

What happens when ethical choices come at the expense of profit?

87/89

Regulation and pc

USNews NEWS

Home / News / Business News / House Approves Antitrust Bill ...

House Approves Antitrust Bill Targeting Big Tech Dominance

The House has approved antitrust legislation targeting the dominance of Big Tech companies by giving states greater power in competition cases and increasing money for federal regulators.

By Associated Press

Sept. 29, 2022

In Its First Monopoly Trial of Modern Internet Era, U.S. Sets Sights on Google

The 10-week trial, set to begin Tuesday, amps up efforts to rein in Big Tech by targeting the core search business that turned Google into a $1.7 trillion behemoth.

Bill Gates, 1998

Source: US News, NYTimes Sep 12, 2023

Across the ocean…

Press release | 20 March 2019 | Brussels

Antitrust: Commission fines Google €1.49 billion for abusive practices in online advertising

E.U. Takes Aim at Big Tech's Power With Landmark Digital Act

The Digital Markets Act is the most sweeping legislation to regulate tech since a European privacy law was passed in 2018.

March 24, 2022

Source: European Commission, NYTimes