This Lecture

• GUIs and JavaFX
• JavaFX Scene Graph Hierarchy
  o Aide: VBox
• Example: ColorChanger
  o EventHandling
  o Private inner classes
  o Random number generation
  o javafx.scene.paint.Colors
• Logical vs. Graphical Containment with JavaFX
• Animation & Timelines
• Example: Clock

What is JavaFX?

• JavaFX is an API (Application Programming Interface) to a graphics+media library
• Simply put, JavaFX is a set of graphics and media packages that enables developers to design, create and test rich and powerful graphical applications for desktop, web and mobile devices
• JavaFX is a successor to the popular SWING library, previously used in CS15 projects to handle graphics!

Creating Applications from Scratch

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

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

Creating GUIs With JavaFX (1/2)

- The main App class for our JavaFX application extends the abstract class javafx.application.Application.
- From now on in CS15, will begin every project by implementing Application class' start method.
- This method is called automatically by JavaFX to launch program.
- Java also creates javafx.stage.Stage that is passed into start method.
- Once shown via the show method, the Stage is effectively a window for the application.

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

Creating GUIs With JavaFX (2/2)

- In order for our application to be more than just a window, must first set the scene before showing the stage.
- javafx.scene.Scene is the top-level container for all UI elements.
  - First instantiate Scene within app class' start method.
  - Then pass that Scene into Stage's setScene(Scene scene) method to set the scene!
- In CS15, only specify 1 Scene – though JavaFX does permit creation of applications with multiple Scenes.
  - Ex: A gaming application where you could select to play either DoodleJump, Tetris or Pacman from the main screen might utilize multiple Scenes – one for each subgame.
- What exactly is a javafx.scene.Scene?

JavaFX Scene Graph Hierarchy (1/3)

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

```
        Scene
          ├── Scene
          └── Scene
            └── Scene
```
You can think of the Scene Graph as a family tree of visual elements.

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

- Each UI component that is added to the Scene Graph as a `Node` gets displayed graphically.

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

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

Constructing the Scene Graph:
- Instantiate root `Node`.

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

- Instantiate root Node
- Then pass into Scene constructor to construct Scene Graph
  - Scene Graph starts off as a single root Node with no children

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

Adding UI elements to the Scene

- 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!

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

10/13/15
Add 3 Buttons to the Scene by adding them as children of the root Node

```
/* within App class */
@Override
public void start(Stage stage) {
    //code for setting root, stage, scene elided
    Button b1 = new Button();
    Button b2 = new Button();
    Button b3 = new Button();
    root.getChildren().addAll(b1, b2, b3);
}
```

What is this unfamiliar double method call notation?

`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!

```
/* within App class */
@Override
public void start(Stage stage) {
    //code for setting root, stage, scene elided
    Button b1 = new Button();
    Button b2 = new Button();
    Button b3 = new Button();
    root.getChildren().addAll(b1, b2, b3);
    root.getChildren().add(b3);
}
```

• Similarly, can remove a UI element by removing it from root's children

```
/* within App class */
@Override
public void start(Stage stage) {
    //code for setting root, stage, scene elided
    Button b1 = new Button();
    Button b2 = new Button();
    Button b3 = new Button();
    root.getChildren().addAll(b1, b2, b3);
    root.getChildren().remove(b3);
}
```

• What if we want to make more complex applications?
• Can add layout Pane as child of root Node, then add more UI elements as child Nodes of this Pane
• This will continue to populate the scene graph!
Populating the Scene Graph

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

```java
/* within App class */
@Override
public void start(Stage stage) {
    //code for setting scene elided
    Button b1 = new Button();
    Button b2 = new Button();
    root.getChildren().add(b1, b2);
    VBox holder = new VBox();
    root.getChildren().add(holder);
}
```

Populating the Scene Graph

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

```java
/* within App class */
@Override
public void start(Stage stage) {
    //code for setting scene elided
    Button b1 = new Button();
    Button b2 = new Button();
    root.getChildren().add(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 (1/2)

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

Removing a Node with children (2/2)

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

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

- Note that while removing a node with no children simply removes that node...
  - `root.getChildren().remove(b2);` to remove second `Button`
- Removing a Node with children removes all of its children as well!
  - `root.getChildren().remove(holder);` makes both `VBox` and `Label` disappear

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 the 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)
  - `root.setAlignment(Pos.TOP_LEFT)` method, passing in a `javafx.geometry.Pos` constant
- Pos options are in the form `VERTICAL_HORIZONTAL`
  - e.g. `Pos.BOTTOM_LEFT` represents positioning on the bottom vertically, right horizontally
  - full list of `Pos` constants can be found here

VBox layout pane (3/5)**

- The following code produces the example on the right:

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

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

• Adding spacing between children

```java
VBox root = new VBox();
Button b1 = new Button("Top");
Button b2 = new Button("Middle");
Button b3 = new Button("Bottom");
root.getChildren().addAll(b1, b2, b3);
root.setSpacing(8);
```

//code for setting the Scene elided

VBox layout pane (5/5)***

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

```java
VBox root = new VBox();
Button b1 = new Button("Top");
Button b2 = new Button("Middle");
Button b3 = new Button("Bottom");
root.getChildren().addAll(b1, b2, b3);
root.setSpacing(8);
root.setAlignment(Pos.TOP_CENTER);
```

//code for setting the Scene elided

CS15 PaneOrganizer Class

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

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

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

• PaneOrganizer will instantiate root Pane, and provide a public getRoot() method that returns this root

  o App class can now access root Pane through PaneOrganizer’s public getRoot() method and pass root into Scene constructor

Pattern

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

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

- Spec: App that contains text reading "CS15 Rocks!" and a Button that randomly changes the text's color with every click

- Useful classes: Stage, Scene, VBox, Label, Button, EventHandler

---

**Process: ColorChanger**

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

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 EventHandler

---

**ColorChanger: App class**

1. When implementing start, we:
   
   A. Instantiate a PaneOrganizer and store it in the local variable organizer

---

**ColorChanger: App class**

1. When implementing start, we:
   
   A. Instantiate a PaneOrganizer and store it in the local variable organizer

   B. Instantiate a new Scene, passing it to the rootPane created above, and set the rootPane to be the root of the Scene

   C. Set the Scene, title the Stage, and show the Stage!
ColorChanger: App class

1. When implementing start, we:
   A. Instantiate a PaneOrganizer and store it in the local variable organizer
   B. Instantiate a new Scene, passing in:
      1. the rootPane, which is accessed through organizer's public getRoot() method
      2. along with desired width and height of Scene
   C. Set the Scene, title the Stage, and show the Stage

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

Process: ColorChanger

1. Create App class that extends javafx.application.Application and implements start (where you set Scene)
2. Create PaneOrganizer class that instantiates root Pane and provides public getRoot() method to return the Pane. In PaneOrganizer, instantiate a Label and Button and add them as children of root Pane
3. Set up a custom EventHandler that changes Label’s color each time Button is clicked, and register Button with this new EventHandler

ColorChanger: Our PaneOrganizer Class

2. In writing our PaneOrganizer class, we:
   A. Instantiate root VBox and store it in instance variable _root

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

ColorChanger: Our PaneOrganizer Class

2. In writing our PaneOrganizer class, we:
   A. Instantiate root VBox and store it in instance variable _root

   ```java
   public class PaneOrganizer {
       private VBox _root;
       public PaneOrganizer() {
           _root = new VBox();
           _root.getChildren().add(new Label("CS15 Rocks!");
       }
   }
   ```
2. In writing our PaneOrganizer class, we:

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

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

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

2. In writing our PaneOrganizer class, we:

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

D. Add Label and Button as children of root

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

Process: ColorChanger

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

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

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

Responding to User Input

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

• We refer to this as event handling

   - A source (Node, such as a Button) generates an Event (such as a mouse click) and notifies all registered EventHandlers

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

- Button click causes Java to generate a javafx.event.ActionEvent
  - ActionEvent is only one of the many javafx.eventTypes that are subclasses of Event
- Classes that implement EventHandler can handle any subclass of Event
  - When a class implements the 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 the <> is known as a "generic parameter" - more on these later!
  - Lets you specialize the interface to deal in all its methods only with a specialized subclass of Event
  - Forces you to replace whatever is inside the literal <> with some subclass of Event, such as ActionEvent, whenever you write a class that implements EventHandler interface (see next slide)

EventHandler (3/3)

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

Back to ColorChanger: Process

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

```java
public class ClickHandler implements EventHandler<ActionEvent> {
    public void handle(ActionEvent e) {
        //code to change _cs15Rocks Label
    }
}
```
3. Defining our custom EventHandler, ClickHandler:

- ClickHandler must listen for the click event and respond to it by changing the color of the “CS15 Rocks!” Label.
- How will the ClickHandler access the Label?
  - Multiple ways to do this: ClickHandler constructor take in a Label as a parameter.
  - This works, but is there a better way?

```java
public class ClickHandler implements EventHandler<ActionEvent> {
    private Label _cs15Rocks;

    public ClickHandler(Label label) {
        _cs15Rocks = label;
    }

    public void handle(ActionEvent e) {
        // code to change _cs15Rocks Label
    }
}
```

Aside: Private Inner Classes (1/2)

- Until now, all classes we have created have been public
  - Live in their own file
  - Can be accessed from within any class
- Now we’ll introduce private inner classes!
  - Useful when there is a class, such as an EventHandler, for which only need to create a single instance, from within a single class
  - Private inner classes have access to the instance variables/methods of class they are contained in (that declared them)
  - Inner classes are a convenient and safe shortcut — don’t require a file.

Aside: Private Inner Classes (2/2)

- Rather than making the ClickHandler class a public class in its own file, we can make it a private inner class of the PaneOrganizer class
- Our ClickHandler will then have access to PaneOrganizer’s _label instance variable
- Can then set _label’s text color from within ClickHandler’s handle(ActionEvent) method, without needing to deal with any unnecessary passing around of references to Label.

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

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

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

```java
private class ClickHandler implements EventHandler<ActionEvent> {
    // code elided for now
}
```
ColorChanger: **ClickHandler** Private Inner Class

3. Defining our custom **EventHandler**, **ClickHandler**:

```
public class ColorChanger {
    private static int _root;
    private static Label _label;

    public PaneOrganizer {
        private Pane _root;
        private Label _label;
    }

    public PaneOrganizer() {
        _root = new Pane();
        _label = new Label("CS15 Rocks!");
        _root.getChildren().addAll(_label);
    }

    public void handle(ActionEvent event) {
        // Implementation elided for now
    }
}
```

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 the display hardware
  - each display pixel has an R, G, and B sub-pixel 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 the specific passed-in Red, Green, and Blue integer values in [0,255]
  - ex: Color.WHITE can be expressed as Color.rgb(255,255,255);

ColorChanger: **Our EventHandler, ClickHandler**

3. Defining our custom **EventHandler**, **ClickHandler**:

```
private class ColorChanger extends PaneOrganizer {
    private Button _btn;

    public ColorChanger() {
        _root = new Pane();
        _label = new Label("CS15 Rocks!");
        _root.getChildren().addAll(_label);
    }

    public void handle(ActionEvent event) {
        // Implementation elided for now
    }
}
```

ColorChanger: Back to our **PaneOrganizer** Class

3. Defining our custom **EventHandler**, **ClickHandler**:

```
private class PaneOrganizer {
    private Pane _root;
    private Label _label;

    public PaneOrganizer() {
        _root = new Pane();
        _label = new Label("CS15 Rocks!");
        _root.getChildren().addAll(_label);
    }

    public void handle(ActionEvent event) {
        // Implementation elided for now
    }
}
```

ColorChanger: Creating custom hardware mixtures of Red, Green and Blue "primaries" generated by the display hardware

- To do so, call setOnAction on btn, passing in an instance of our ClickHandler
The Whole App: ColorChanger

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

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

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

```
public class ClickHandler implements EventHandler<ActionEvent> {
    public void handle(ActionEvent event) {
        int red = (int) (Math.random() * 256);
        int green = (int) (Math.random() * 256);
        int blue = (int) (Math.random() * 256);
        Color customColor = Color.rgb(red, green, blue);
        _label.setTextFill(customColor);
    }
}
```

Putting it all together

```
Logical vs. Graphical Containment

• Graphically, the VBox is contained within the Scene, but logically, i.e., programmatically, the VBox is contained within the PaneOrganizer
• Graphically, the Button and Label are contained within the VBox, but logically, the Button and Label are contained within the PaneOrganizer
• Logical containment is based on where objects are instantiated, while graphical is based on JavaFX elements being added to other JavaFX elements via the getChildren.add(…) method
```

```
Animation – Change Over Time

• Suppose we have an alien Shape we would like to animate (e.g., make him move across the screen)
• As in film and video animation, we can create apparent motion with many small changes in position
• If we move fast enough and in small enough increments, we get smooth motion
• Same goes for size, orientation, shape change, etc…
• How to orchestrate a sequence of incremental changes
    - By coordinating them with a Timeline where change happens at defined instants
```

```
```
Introducing Timelines (1/2)

KeyFrame represents a set of values (typically graphics-related) which is subject to (incremental) change when the KeyFrame’s time is up (specified by duration) – EventHandler makes the appropriate changes.

We don’t need the full generality, and can do simple animation using a single KeyFrame that is executed a fixed number of times or an indefinite number of times; each time the EventHandler is called, it makes incremental changes to the time-varying variables (e.g. (x,y) position of a character).

Using JavaFX Timlines (1/2)

- `javafx.animation.Timeline` is used to sequence one or more KeyFrames, and optionally to run through them cyclically
  - Each KeyFrame lasts for its entire duration without making any changes, until its time interval ends and the EventHandler is called to make updates
  - When we instantiate a KeyFrame, we pass:
    - A `Duration` (e.g. `Duration.seconds(0.3)` or `Duration.millis(300)`), which defines the time each KeyFrame lasts
    - An EventHandler that defines what should occur upon completion of each KeyFrame

Using JavaFX Timlines (2/2)

- We then pass a new KeyFrame into the Timeline
- After we instantiate our Timeline, we must set its CycleCount property
  - This defines the number of cycles in the Animation
  - We will set the cycle count to `Animation.INDEFINITE`, which will let the Timeline run forever or until we explicitly stop it
- In order for the Timeline to work, we must then call `Timeline.play()`
Another JavaFX App: **Clock**

- Simple example of a discrete (non-smooth) animation
- Specifications: App should display the current date and time, updating every second

Useful classes:
- java.util.Date
- javafx.util.Duration
- javafx.animation.KeyFrame
- javafx.animation.Timeline

**Process: Clock**

1. Write App class that extends javafx.application.Application and implements start
2. Write a PaneOrganizer class that instantiates the root node and returns its public getRoot() method. Instantiate a Label and add it as the root node's child. Factor out code for Timeline into its own method.
3. In our own setupTimeline(), instantiate a KeyFrame passing in a Duration and an instance of TimeHandler (defined later). Then instantiate a Timeline, passing in our KeyFrame, and play the Timeline.
4. Write a private inner TimeHandler class that implements EventHandler - it should know about a Label and update its text every ActionEvent

**Clock: App class**

```
public class App extends Application {
    private PaneOrganizer organizer;

    @Override
    public void start(Stage stage) {
        PaneOrganizer organizer = new PaneOrganizer();
        Scene scene = new Scene(organizer.getRoot(), 200, 200);
        stage.setScene(scene);
        stage.setTitle("Color Changer");
        stage.show();
    }
}
```
Clock: App class

**Note: Exactly the same process as in ColorTextApp**

1. Instantiate a PaneOrganizer and store it in the local variable organizer.
2. Instantiate a Scene, passing in organizer.getRoot(), desired width of the Scene, and desired height of the Scene as parameters.
3. Set the Scene, set the Stage’s title, and show the Stage!

```java
public class App extends Application {
    @Override
    public void start(Stage stage) {
        PaneOrganizer organizer = new PaneOrganizer();
        Scene scene = new Scene(organizer.getRoot(), 200, 200);
        stage.setScene(scene);
        stage.setTitle("Color Changer");
        stage.show();
    }
}
```

**Note:** Exactly the same process as in ColorTextApp.

---

Clock: Our PaneOrganizer Class

A. In the PaneOrganizer class’ constructor, instantiate a root VBox and set it as the return value of a public getRoot() method.

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

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

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

B. Instantiate a Label and add it to the list of the root node’s children.

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

    public PaneOrganizer() {
        _root = new VBox();
        _label = new Label();
        _root.getChildren().add(_label);
    }

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

---

Process: Clock

1. Write App class that extends javafx.application.Application and implements start.
2. Write a PaneOrganizer class that instantiates root node and returns it in a public getRoot() method. Instantiate a Label and add it as root node’s child. Factor out code for Timeline into its own method.
3. In our own setupTimeline(), instantiate a KeyFrame passing in a Duration and an instance of TimeHandler (defined later). Then instantiate a Timeline, passing in our KeyFrame, and play the Timeline.
4. Write a private inner TimeHandler class that implements EventHandler - it should know about a Label and update its text on every ActionEvent.

---

Clock: Our PaneOrganizer Class

A. In the PaneOrganizer class’ constructor, instantiate a root VBox and set it as the return value of a public getRoot() method.

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

    public PaneOrganizer() {
        _root = new VBox();
        _label = new Label();
        _root.getChildren().add(_label);
    }

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

B. Instantiate a Label and add it to the list of the root node’s children.

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

    public PaneOrganizer() {
        _root = new VBox();
        _label = new Label();
        _root.getChildren().add(_label);
    }

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

A. In the PaneOrganizer class' constructor, instantiate a root VBox and set it as the return value of a public getRoot() method

B. Instantiate a Label and add it to the list of the root node's children

C. Call setupTimeline(); will define this method next!!

Clock: PaneOrganizer class - setupTimeline()

Within setupTimeline():

A. Instantiate a KeyFrame

public void setupTimeline()
{
    KeyFrame kf = new KeyFrame(Duration.seconds(1),
    new Timeline());
}

Process: Clock

1. Write an App class that extends javafx.application.Application and implements start

2. Write a PaneOrganizer class that instantiates the root node and returns its public getRoot() method.

3. Instantiate a Label and add it as the root node's child.

4. Factor out code for Timeline into its own method.

5. In setupTimeline(), instantiate a KeyFrame passing in a Duration and an instance of TimeHandler (defined later). Then instantiate a Timeline, passing in our KeyFrame, and play the Timeline.

6. Write a private inner TimeHandler class that implements EventHandler - it should know about a Label and update its text on every ActionEvent

Clock: PaneOrganizer class - setupTimeline()

Within setupTimeline():

A. Instantiate a KeyFrame

public void setupTimeline()
{
    KeyFrame kf = new KeyFrame(Duration.seconds(1),
    new Timeline());
}

Clock: PaneOrganizer class - setupTimeline()

Within setupTimeline():

A. Instantiate a KeyFrame

public void setupTimeline()
{
    KeyFrame kf = new KeyFrame(Duration.seconds(1),
    new Timeline());
}

Clock: PaneOrganizer class - setupTimeline()

Within setupTimeline():

A. Instantiate a KeyFrame

public void setupTimeline()
{
    KeyFrame kf = new KeyFrame(Duration.seconds(1),
    new Timeline());
}
Within `setupTimeline()`:

A. Instantiate a KeyFrame

B. Instantiate a Timeline, passing in our new KeyFrame

C. Set the CycleCount to INDEFINITE

D. Play, i.e. start the Timeline

Process: Clock

1. Write an `App` class that extends `javafx.application.Application` and implements `start`

2. Write a `PaneOrganizer` class that instantiates the root node and returns its root node in a `public void setupTimeline()` method. Instantiate a `Label` and add it as the root node's child.

3. In `setupTimeline()`, instantiate a `Label` and pass in a `Duration` and an instance of `TimeHandler` (defined later). Then instantiate a `Timeline`, passing in our KeyFrame, and play the Timeline.

4. Write a private inner `TimeHandler` class that implements `EventHandler` — it should know about a `Label` and update its text on every `ActionEvent`
A. The last step is to create our TimeHandler and implement handle(), specifying what to occur at the end of each KeyFrame – called automatically by j-fx.

B. java.util.Date represents a specific instant in time. Date is a representation of the time, to the nearest millisecond, at the moment the Date is instantiated.

C. Because our Timeline has a duration 1 second, each second new Date will be generated, converted to a String, and set as the _label's text. This will appropriately update the Label with correct time every second!
Announcements (1/2)

- TASafeHouse is due tonight at 11:59PM
  - late handin is Saturday 10:00 pm. Handins after this deadline will receive no credit
- 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
- We have posted a JavaFX guide to our website
  - Read this if you’d like a more complete overview of JavaFX and extra examples
  - Note: a couple examples may seem confusing, but don’t worry, any unfamiliar syntax will all be covered in lecture by next Thursday!

Announcements (2/2)

- We are running another completely optional Q&A group tutoring session Saturday 1pm-2:30pm in CIT 201
  - We’ll send an email on what exactly we’re covering
- Cartoon goes out on Tuesday – happy long weekend!