Lecture 9
Intro to JavaFX

This Lecture
• GUIs and JavaFX
  • JavaFX Scene Graph Hierarchy
    o Aside: VBox
  • Example: ColorChanger
    o Event Handling
    o Private inner classes
    o Random number generation
    o javafx.color.Color
  • Logical: Graphical Containment with JavaFX
    o Animation & Timelines
    • Example: Clock

Pixels and Coordinate System
• Screen is a grid of pixels (tiny squares, each with RGB components)
• Cartesian plane with:
  o origin in upper-left corner
  o x-axis increasing left to right
  o y-axis increasing top to bottom
  o corresponds to English writing order
• Each graphical element positioned at specific pixel

What is JavaFX?
• Usually don’t want to program at the pixel level
• JavaFX is an API (Application Programming Interface) to a graphics + media library: a collection of useful classes, interfaces, and their methods (with suitable documentation) — no internals!
• Simply put, 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 a successor to the popular SWING library, previously used in CS15 projects to handle graphics!
• Shout out to 2016 CS15 TA Montana’s mom who worked on JavaFX!

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, we’ll begin every project by implementing the Application class’ start method.
- Java also creates javaFX.stage.Stage that is passed into the 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 Doodadump, Tetris or Platform from the main screen to 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.

JavaFX Scene Graph Hierarchy (2/3)

- 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.

JavaFX Scene Graph Hierarchy (3/3)

- 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 Panes have different built-in layout capabilities to allow easy positioning of UI elements—see inheritance tree below for flavors.
- For now, use a VBox as the root of the Scene—more on VBox later.
Constructing the Scene Graph

- Instantiate root Node

public class App extends Application {
    @Override
    public void start(Stage stage) {
        //code for setting root, stage, scene elided
    }
}

Adding UI elements to the Scene (1/2)

- How do we add more Nodes to the Scene Graph?
- Adding UI elements as children of root Node will add them to Scene and make them appear on the Stage!
- Calling getChildren() method on a Node returns a list of that Node’s children
- By adding/removing nodes from that list of children, we can add/remove Node from the Scene Graph!

Adding UI elements to the Scene (2/2)

- Recap: getChildren() returns a List of child Nodes
  - In example on right, root.getChildren() returns a list holding three Buttons (assuming we created them previously - next slide)
- To add a Node to this list of children, call add(Node node) on that returned list!
  - Can also use addAll(Nodes, node1, node2, ...)
    - which takes in any number of Nodes
    - Allowing a number of arguments is a new capability of parameter list
- To remove a Node from this list of children, call remove(Node node) on that returned list!

root.getChildren().add(...) in action

- Add 3 Buttons to the Scene by adding them as children of the root Node (empty before this)
- Order matters here!
- Within App class:
  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().add(b1, b2, b3);
  }

Avoid adding method call children?
- root.getChildren().add() returns a list of root’s children. Rather than storing that returned list in a variable and calling add(...) on that variable, we can directly call add(...) on the returned list of children!
Populating the Scene Graph (1/5)

- Similarly, can remove a UI element by removing it from root's children
  - Note: The order of the children doesn't matter when removing elements

```java
private 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);
}  
```

Populating the Scene Graph (2/5)

- Similarly, can remove a UI element by removing it from root's children
  - Note: The order of the children doesn't matter when removing elements

```java
private 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);
  root.getChildren().remove(b3);
}  
```

Populating the Scene Graph (3/5)

- 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!

```java
private 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);
}  
```

Populating the Scene Graph (4/5)

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

```java
private void start(Stage stage) {
  // code for setting scene elided
  Button b1 = new Button("Button 1");
  Button b2 = new Button("Button 2");
  VBox holder = new VBox();
  Label text = new Label("I live in the VBox!");
  holder.getChildren().add(text);
}  
```

Populating the Scene Graph (5/5)

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

```java
private void start(Stage stage) {
  // code for setting scene elided
  Button b1 = new Button("Button 1");
  Button b2 = new Button("Button 2");
  VBox holder = new VBox();
  Label text = new Label("I live in the VBox!");
  holder.getChildren().add(text);
}  
```

Removing a Node with children (1/2)

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

Clicker Question

Which of the following correctly removes `removeLabel` from the VBox holder?

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

VBox layout pane (1/5)***

- So what exactly is a VBox?
  - **VBox layout Pane** creates an easy way for arranging a series of **children** in a **single vertical column**
  - We can customize vertical spacing between children using VBox's `setSpacing(double)` method
    - The larger the `double` passed in, the more space between the child UI element

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
    - Pos options are in the form **Pos.VERTICAL_HORIZONTAL**
      - e.g. `Pos.BOTTOM_RIGHT` 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);
  ```
  - Scene scene = new Scene(root, 200, 200);
  - stage.setScene(scene); // Sample VBox

Why ALL_CAPS notation?

It is a "symbolic constant" with pre-defined meaning.
**CS15 PaneOrganizer Class (1/2)**

- 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
  - App class can now access root Pane through PaneOrganizer's public getRoot() method and pass root into Scene constructor
- We'll do this together in a sec!

**CS15 PaneOrganizer Class (2/2)**

**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 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
   B. Instantiate a new Scene, passing in:
      - `rootPane`, accessed through organizer's `getRoot()` method
      - along with desired width and height of scene
   C. Set the Scene, the Stage, and show the Stage

**ColorChanger: Our PaneOrganizer Class**

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

**Process: ColorChanger**

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

**ColorChanger: Our PaneOrganizer Class**

2. In writing our PaneOrganizer class, we:
   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 with App class
EventHandler: Our PaneOrganizer Class

2. In writing our PaneOrganizer class, we
   - Instantiate Label and Button, passing in String representations of text we want displayed
     - Label is an instance variable
     - Button is a local variable that is only needed to access it within constructor
   - Button changes children of root
     - btn = new Button("Random Color");
     - _label = new Label("CS15 Rocks!");
   - Add Label and Button as children of root

ColorChanger: Our PaneOrganizer Class

2. In writing our PaneOrganizer class, we
   - Instantiate Label and Button, passing in String representations of text we want displayed
   - Label is an instance variable
   - Button is a local variable that is only needed to access it within constructor
   - Button changes children of root

ColorChanger: Our PaneOrganizer Class

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

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
  - An EventHandler is an interface, so all classes that implement it must implement handler(ActionEvent event) method, which defines response to event

EventHandlers (1/3)

- Button click causes JavaFX to generate a javafx.event.ActionEvent
  - ActionEvent is an event of type Event
  - Classes that implement EventHandler can polymorphically handle any subclass of Event
  - When a class implements EventHandler interface, it must specify what type of event it should handle
  - How do we do this?
  - EventHandler interface declared as: public interface EventHandler extends Event

EventHandlers (2/3)

- We want our custom EventHandler (call it ClickHandler) to handle an ActionEvent, so ClickHandler will implement "EventHandler<ActionEvent>" interface

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
3. In PaneOrganizer, instantiate a Label and Button and add them as children of root Pane
4. Set up a custom EventHandler that changes Label’s color each time Button is clicked, and register Button with this new EventHandler
EventHandlers (3/3)

- Our ClickHandler must implement the handle(ActionEvent) method of the EventHandler interface, which will specify the response to the ActionEvent (in this case, a click).
- For now, we'll make the EventHandler base class (without parameters).
- To tell this new ClickHandler to listen for the Button's ActionEvent, register Button with the ClickHandler by calling btn.setOnAction(ClickHandler).
- In this case, passing in an instance of our ClickHandler class.
- The mechanics of changing the color of a Label's text color.

ClickHandler: Private Inner Classes

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

Aside: Private Inner Classes (1/2)

- Until now, all classes we have created have been public:
  - can be accessed from any class.
  - Now we'll introduce private inner classes:
  - useful when there is a class, which is 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.

ColorChanger: Private Inner Class

3. Defining our custom EventHandler, ClickHandler:
   - In order to make ClickHandler a private inner class of PaneOrganizer class, we simply declare it as a private class and place it within the brackets of public PaneOrganizer class.
ColorChanger: Private Inner Class

```java
public class ClickHandler implements EventHandler<ActionEvent> {
    private Label _label;
    private VBox _root;

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

3. Defining our custom Event Handler, ClickHandler:

- Implement `handle` method
- Use `new Button(...)`
- Use a `ClickHandler` instance

Generating `javafx.scene.paint.Color`

- We can generate most colors of visible color spectrum by additive mixtures of Red, Green, and Blue "primaries" generated by display hardware
- Each display pixel has an R, G, and B sub-pixel to do this color mixing

Hardware mixtures of Red, Green, and Blue "primaries" generated by display hardware

ColorMixing.png

ColorChanger: Our EventHandler, ClickHandler

3. Defining our custom Event Handler, ClickHandler:

- `Math.random()` returns a random double between 0 inclusive and 1 exclusive
- Defining the variables `R`, `G`, and `B`
- Use these ints as `Red`, `Green`, and `Blue`
- Our custom `Color` class

The Whole App: ColorChanger

```java
public class ColorChanger {
    public void start(Stage stage) {
        // Implementation
    }
}
```

Putting It All Together

```
Application
    
    PaneOrganizer
        Button
            ClickHandler
            Label
                setTextFill(customColor)
```

10/10/16
Logical vs. Graphical Containment/Scene Graph

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

Announcements (1/2)

- TAs for the course have changed:
  - Early handin is Friday 10:00pm
  - On-time handin is Sunday 11:59pm
- 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 be covered in lecture by next Thursday!

Announcements (2/2)

- Lab 3 goes out today!
  - You must get Lab 2 checked off by the end of your lab this week to get credit.
- Review Sessions are this Thursday and Sunday.