Chapter 15: JavaFX

CS 2070

March 20, 2018

Introduction to GUI Concepts

Stages and Scenes

Scene Graphs and Scene Graph Nodes

Panes and Component Layout

Events and Event Handling

Determining the Target of an Event

Radio Buttons

Displaying Images and Inputting Text

Introduction to GUI Concepts

Introduction to GUI Concepts

- GUI stands for “Graphical User Interface”.
- GUI programs are graphical components, such as buttons, textfields, menus, and scrollbars.
- Users employ a mouse and keyboard to interact with the program.

Event Driven Programming

Events occur when a user interacts with the program through a GUI component. There are usually two varieties.

- Mouse clicks.
- Key presses.

Event Handling

When an event occurs on a component, the program needs to handle it by calling an appropriate method in an event-handler object that has previously been attached to that component.

General Format for GUI programs.

- Create the user interface and arrange components.
- For each component, identify the events of that component that your program needs to handle.
- Write the handlers to those events.
- Display the user interface.

JavaFX

- JavaFX is a the latest class libraries for creating GUI programs in Java.
- JavaFX can be used to develop GUI applications that run on the desktop, inside of a web browser, and on mobile devices such as tablets and smart phone
- JavaFX is intended to replace the older and better known Swing library.
  - You've used Swing before. JOptionPane is part of Swing.

Stages and Scenes

GUI Programming with JavaFX

To create a JavaFX program, we create a subclass of `Application`.

```
public class JavaFXApplication1
    extends Application {

    @Override
    public void start(Stage stage) {
        // Build User Interface
        // and attach event handlers
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

An aside: What's subclass?

- One of your recent homework assignment required that you make a class.
- A subclass way of creating new classes with new features based on existing classes without having to rewrite an existing class.
- The term for this feature is **inheritance**.
- Inheritance promotes code reuse.
- The Java keyword to create a subclass is **extends**.
  - Look for this keyword in the previous slide. What are we extending?

The launch method.

- The launch() method is a static method inherited from Application. It should be called by the main() method.
- The launch() method sets up the JavaFX Application object, creates a Stage object, and calls the start() method of the Application object with the stage object as parameter.

The start() Method

- Receives a Stage object as parameter
- Creates a Scene object that consists of a hierarchy of UI components and sets the scene onto the stage object
- Sets event handlers on some of the components
- Shows the stage to the user

Scene Graphs and Scene Graph Nodes

Scene Graphs and Scene Graph Nodes

- The totality of UI components appearing on the screen of a JavaFX application form a scene
- The UI components are nested, forming a hierarchical tree structure called a scene graph
- Each component that is part of a scene graph is called a scene graph node

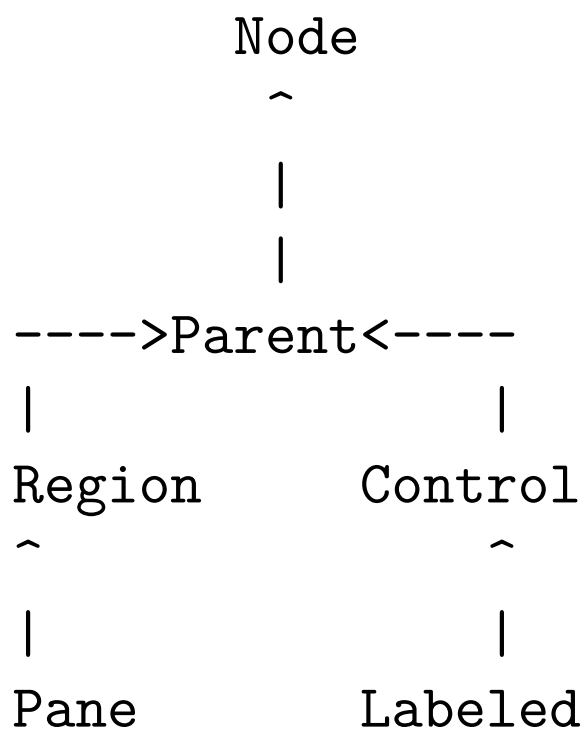
The Node Class

- The Node class is the superclass of all classes of that describe a component that can appear on the screen as part of a scene
- Examples of subclasses of Node are Button, TextField, Label, and various types of panes
- A pane is a UI component that can contains other UI components and arranges them according to some layout discipline

Branch Nodes and Leaves

- In a scene graph, a node that contains other nodes is called a **branch node**
- A scene graph node that does not contain other nodes is called a **leaf**
- A branch node is the **parent** of the nodes it contains; the contained nodes are called **children** of the branch node

Node and its Subclasses



Node and its Subclasses

- Node: A UI component with a visual representation on the screen
- Parent: A UI component that can contain other UI components (can have children)
- Region: A container that can layout its children and have its appearance styled using CSS
- Pane: A container that allows programmers to add and remove child nodes
- Control: A UI component that can be used to interact with and exchange information with the user (e.g. TextField, Button, Label, etc)

Creating a Scene

- All the UI components that will appear on the screen are nested in such a way that they form a hierarchical tree structure called the scene graph
- The top-most container is called the root of the scene graph; it must be an instance of the Parent class
- The root of the scene graph is used to create a scene via one of the constructors for the Scene class:
  - Scene(Parent root, double width, double height)
  - Scene(Parent root)
- The created scene object is set onto the stage and the stage is shown:
  - Parent root = new Label("Hello World");
  - Scene scene = new Scene(root);
  - stage.setScene(scene);
  - stage.show();

Example JavaFX Program

```
public class JavaFXHelloWorld extends Application {
    @Override
    public void start(Stage stage) {
        // Create label
        Label label = new Label("Hello World!");
        // Set label as root of scene graph.
        Scene scene = new Scene(label , 300, 80);
        stage.setScene(scene);
        // Set stage title and show the stage.
        stage.setTitle("Hello World!");
        stage.show();
    }
}
```

Panes and Component Layout

Panes and Component Layout

- Pane objects keep their child nodes in a collection of type ObservableList which has two method that can be used to add nodes
  - boolean add(Node child)
  - boolean addAll(Node... children)
- The Pane class exposes this list of child nodes through a method
  - ObservableList getChildren()

One vertical column

The Vbox Pane arranges is children in one vertical column:

```
public void start(Stage stage) {
  Button b1 = new Button("One");
  Button b2 = new Button("Two");
  Button b3 = new Button("Three");

  VBox vPane = new VBox();
  vPane.getChildren().addAll(b1, b2, b3);

  stage.setScene(new Scene(vPane));
  stage.show();
}
```

One vertical column with spacing

A Vbox(double spacing) constructor inserts vertical spacing between its children:

```
public void start(Stage stage) {
  Button b1 = new Button("One");
  Button b2 = new Button("Two");
  Button b3 = new Button("Three");

  VBox vPane = new VBox(10);
  vPane.getChildren().addAll(b1, b2, b3);

  stage.setScene(new Scene(vPane));
  stage.show();
}
```

Alignment

By default, children of Vbox huddle together in the top left corner of the pane
This default alignment can be changed by calling the Vbox method

```
void setAlignment(Pos value)
```

and specifying a Pos enumerartion value as parameter

Pos Values

Pos is an enumeration type whose values specify vertical and horizontal alignment of content:

`TOP_LEFT`	`TOP_CENTER`	`TOP_RIGHT`
`CENTER_LEFT`	`CENTER`	`CENTER_RIGHT`
`BOTTOM_LEFT`	`BOTTOM_CENTER`	`BOTTOM_RIGHT`

- You can set alignment on a pane to CENTER:
  - vPane.setAlignment(Pos.CENTER);
- The Label class has a similar method for setting alignment of content:
  - myLabel.setAlignment(Pos.CENTER);

Hbox Pane

The Hbox Pane is similar to Vbox, except it lays out its children in a single horizontal row:

```
Button b1 = new Button("One");
Button b2 = new Button("Two");
Button b3 = new Button("Three");

HBox hPane = new HBox(10);
hPane.getChildren().addAll(b1, b2, b3);
```

Margin and Padding

- Margin and Padding are used to achieve spacing and give the UI a pleasing look
- Margin is the spacing around the outside border of a node
- Padding is the spacing inside the border of a node: padding surrounds the node's content and sets it off from the node's border

Margin and Padding

- Margin and Padding are specified by objects of type Inset:
  - Insets(double top, double right, double bottom, double left)
  - Insets(double width)

Setting Margin and Padding

- Pane has method for setting the padding around its content:
  - void setPadding(Insets value)
- Pane has a static method that sets the margin around a a specific child node.
  - static void setMargin(node child, insets value)

Effect of Margin, Padding and Alignment

```
HBox hPane = new HBox();
hPane.getChildren().addAll(b1, b2, b3);

hPane.setPadding(new Insets(10));

HBox.setMargin(b2, new Insets(0, 20, 0, 10));

hPane.setAlignment(Pos.CENTER);
```

Nested Layouts

- You can achieve the look you want by nesting different types of panes
- The above UI uses a Vbox with center alignment containing a label and a Hbox, also with center alignment

Events and Event Handling

Events and Event Handling

- An event is an occurrence within a program that requires a response
- An event handler is an object containing a method that is called to respond to an event
- Events occur on UI components
- Event handlers are set on UI components to respond to events that occur on those components

ActionEvent

- The ActionEvent class describes certain types of events where the user is expecting the program to respond by performing some sort of action
- An ActionEvent is generated by Button when the user clicks a button
- An ActionEvent is generated by TextField when the user types ENTER in the text field Handling ActionEvent
- An event handler for ActionEvent is an object of a class that implements the interface EventHandler
- This interface has a single abstract method void handle(ActionEvent evt)

Code.

```
class SimpleHandler implements EventHandler<ActionEvent> {
    public void handle(ActionEvent event) {
        JOptionPane.showMessageDialog(null, "Hello World!")
    }
}
```

Setting an Event Handler on a Component

Here is how to create and set an event handler on a button:

```
public void start(Stage stage) {
    // Button in HBox
    Button b1 = new Button("Click Me");
    HBox hPane = new HBox();
    hPane.setAlignment(Pos.CENTER);
    hPane.getChildren().add(b1);

    // Create scene and show on stage
    stage.setScene(new Scene(hPane));
    stage.show();

    // Set handler on button
    b1.setOnAction(new SimpleHandler());
}
```

Passing Information to Event Handlers

- Imagine a program that displays a label with a number that starts at 0, together with a button
- Every time the button is clicked, the number on the label is incremented by 1

Passing Information to an Event Handler

Create the Label and Button.

```
Label label = new Label("0");
Button button = new Button("Click");
```

- and add them to a Vbox pane
- Create an event handler class called ClickHandler
- The click handler object will need access to the label so it can increment. The label will be passed to handler when it is being created: new ClickHandler(label);

Passing Information to an Event Handler

```
public void start(Stage stage) {
    // Create label, button, attach handler to button.
    Label label = new Label("0");
    Button button = new Button("Click");
    button.setOnAction(new ClickHandler(label));

    // Add the label and button to a pane.
    VBox pane = new VBox(10);
    pane.setAlignment(Pos.CENTER);
    pane.getChildren().addAll(label, button);

    // Set up the stage.
    stage.setScene(new Scene(pane, 200, 80));
    stage.setTitle("Click Count");
    stage.show();
}
```

The ClickHandler Class

The ClickHandler class has a Label field to hold the information passed to its constructor:

```
 // Reference to label that will be updated
private Label rLabel;
public ClickHandler(Label cParamLabel) {
    rLabel = cParamLabel;
}
```

The ClickHandler handle() Method

The handle()method has access to the label so it can retrieve the number from the label, increment it, and set it back onto the label:

```
class ClickHandler implements EventHandler<ActionEvent> {
    private Label rLabel;
    public ClickHandler(Label cParamLabel) {
        rLabel = cParamLabel;
    }
    @Override
    public void handle(ActionEvent event) {
       int count = Integer.parseInt(rLabel.getText());
       count ++;
       rLabel.setText(String.valueOf(count));
    }
}
```

Inner classes as Event Handlers

- If you define the handler as a local inner class, it will automatically have access to all local variables that are effectively final
- Local inner classes are convenient to use as handlers because you do not need to use constructor parameters to pass information
- Using a separate class for the handler often makes for cleaner and more reusable code

Using an Inner Class as Handler

```
public void start(Stage stage) {
    // Create label, button, and attach event handler to th
    Label label = new Label("0");
    Button button = new Button("Click");

    // Define an inner class to use as event handler.
    class ClickHandler implements EventHandler<ActionEvent>
       public void handle(ActionEvent event) {
           int count = Integer.parseInt(label.getText());
           count++;
           label.setText(String.valueOf(count));
       }
    }
    // Create a handler based on the inner class and set or
    button.setOnAction(new ClickHandler());
    // More code ....
}
```

Anonymous Local Inner Classes

- In Java, you can create on object of a class that implements an interface without defining the class and giving it a name.
- Such a class (with no name) is called an anonymous class.
- You instantiate an object of such a class by specifying the interface, providing definitions for the methods in the interface, and using the new operator to instantiate the object

Code.

```
EventHandler <ActionEvent> handler = null;
handler = new EventHandler<ActionEvent> () {
    public void handle(ActionEvent evt) {
      // handler logic goes here
    }
};
button.setOnAction(handler);
```

Lambda Expressions for Event Handling

You can also use a lambda expression for an event handler

```
EventHandler <ActionEvent> handler = null;
handler = evt-> {
   // handler logic goes here
};
button.setOnAction(handler);
```

Lambda expressions are short hand for objects of local anonymous inner classes, so any variable accessed by a lambda expression must be effectively final

Lambda Expressions for Event Handling

```
public void start(Stage stage) {
    // Create label, button
    Label label = new Label("0");
    Button button = new Button("Click");

    // Use a lambda expression for the event handler.
    button.setOnAction(
            event -> {
               int count = Integer.parseInt(label.getText()
               count++;
               label.setText(String.valueOf(count));
            });
    //  additional code ....
}
```

Determining the Target of an Event

The Target of an Event

- When an event occurs and the event handling method is called, the component on which the event occurred is called the target of the event
- The ActionEvent parameter evt passed to the handle() method can be used to identify the event target by calling the instance method `evt.getTarget()`

Using a Single Handler on Multiple Components

We often want to use the same handler to respond to the same event on several components of the same type:

```
EventHandler<ActionEvent> handler1= evt -> {
   // handler logic here
};
button1.setOnAction(handler1);
button2.setOnAction(handler1);
```

When button1 is clicked and handler1's handle() with parameter evt, then evt.getTarget() will return a reference to button1. Similarly for button2. Thus the handle() method can distinguish which button is "calling"

The EventTarget Interface

- Any component that can generate an event implements the EventTarget interface
- The evt.getTarget()method returns a reference to EventTarget:

Code.

```
EventTarget getTarget()
```

Typical use of getTarget() casts the returned object to a known class type, for example

```
public void handle(ActionEvent evt) {
  Button b = (Button) evt.getTarget();
  // Use b
}
```

Determining the Event Target

- Consider a JavaFX program that displays two buttons on a stage
- The title of the stage is the text of whichever button was last clicked

The Event Target Program

```
public void start(Stage stage) {
    Button button1 = new Button("One");
    Button button2 = new Button("Two");

    // Create the event handler using a lambda expression.
    EventHandler<ActionEvent> handler = event -> {
        Button clickedButton = (Button) event.getTarget();
        String newTitle = clickedButton.getText();
        // set the new stage title.
        stage.setTitle(newTitle);
    };

    // Set the same event handler to BOTH buttons.
    button1.setOnAction(handler);
    button2.setOnAction(handler);
    // More code  ...
}
```

Radio Buttons

Radio Buttons

- Radio buttons are used to select a single option from a group of choices
- A radio button becomes selected when it is clicked on
- Radio buttons are typically used in groups, with each radio button corresponding to a single choice in an associated group of choices
- An object called a toggle group is used to manage the radio buttons in a single group, ensuring that at most one of them is selected at any time

RadioButton Constructors and Methods

- RadioButton()
- RadioButton(String text)
- boolean isSelected()
- void setSelected(boolean value)
- void setToggleGroup(ToggleGroup value)

Programming with Radio Buttons

- Consider a program that allows the user to select one of a group of choices and click a button to display the currently selected choice in a label Radio Button Demo Program
- First, create a Vbox with a gray border. This will be used to hold the radio buttons. The gray border is a visual cue that the radio buttons form a group:

Code.

```
// Vertical Box to hold the radio buttons.
VBox radiosBox = new VBox(10);
radiosBox.setPadding(new Insets(10, 10, 10, 10));
// Set a gray border around the radio button box.
radiosBox.setStyle("-fx-border-color: gray;");
```

Radio Button Demo Program

Create an array of strings to use as text for the radio buttons, create a toggle group object, and then create an array of radio buttons:

```
String [] optionLabels =
    {"Walk", "Drive", "Take Public Transportation"};

// Create a toggle group, and the array of radioButtons
ToggleGroup radiosGroup = new ToggleGroup();
RadioButton [] radioButtons =
          new RadioButton[optionLabels.length];

for (int k = 0; k < radioButtons.length; k++) {
   radioButtons[k] = new RadioButton(optionLabels[k]);
   radioButtons[k].setToggleGroup(radiosGroup);
}
```

Radio Button Demo Program

Add the radio buttons to the Vbox and select the first radio button

```
radiosBox.getChildren().addAll(radioButtons);
radioButtons[0].setSelected(true);
```

Construct the top-level box that will be the root of the scene graph

```
VBox topLevelBox = new VBox(10);
topLevelBox.setAlignment(Pos.CENTER);
topLevelBox.setPadding(new Insets(10,50,10,50));
```

The Radio Demo Program

Create the show selection Button and the label to display the selected option and add them to the top-level box

```
Button showSelectionButton =
          new Button("Show Selection");
Label selectionLabel = new Label();
topLevelBox.getChildren().
           addAll(radiosBox,showSelectionButton,
           selectionLabel);
```

Radio Button Demo Program

Finally, create an event handler for the button. The handler determines the selected radio button and sets its text as the text of the selection label

```
// Set the handler for the show selection button.
EventHandler<ActionEvent> handler = event -> {
    for (RadioButton rb : radioButtons) {
        if (rb.isSelected()) {
            selectionLabel.setText(rb.getText());
            return;
        }
    }
};
showSelectionButton.setOnAction(handler);
```

Check Boxes

- Check boxes are a used to select any number of options from a group of options
- Programming with check boxes is like programming with radio buttons, except you do not need to use a toggle group

Displaying Images and Inputting Text

Displaying Images

- Using images requires the use of the Image and ImageView classes
- Image is used to create an in-memory representation of an image.
- The image may be in an InputStream object, or may be in a location online (specified by a URL string), or on the local file system (specified by a pathname string)
- An Image object is not a JavaFX node, so cannot be displayed on the screen
- ImageView is a subclass of Node: it is used to wrap Image objects for screen display

The Image Class and ImageView Classes

- Image(InputStream stream): This constructor creates an image from an input stream, for example
  - new Image(new FileInputStream("bobross.jpg");
- Image(String location): Creates an image by fetching content from a local file system location or from a URL, for example
  - new Image("c:\temp\images\tiger.jpg");
- ImageView(Image image): Creates an ImageView object by wrapping an in-memory Image
- ImageView(String location): Fetches content from the given location, internally creates an Image object, and wraps it.
- ImageView(): creates an ImageView object without an Image. The image can be set with the setImage(Image im) method.

The TextField Control

- The TextField control allows the user to enter a single line of input
- You can create TextField objects using the following constructors
  - TextField()
  - TextField(String text)

TextField Methods

- void setText(String text)
- String getText()
- void setEditable(boolean value)
- boolean isEditable()
- void clear()
- void setPrefColumnCount(int count)

Working With TextField Controls

- Most of the time, we want to process the content of the TextField only after the user presses the ENTER key, and not after every character typed
- When the user types ENTER, the text field will fire an ActionEvent
- We can handle these events by setting an ActionEvent handler on the text field:

Code.

```
EventHandler<ActionEvent> handler =  ....;
myTextField.setOnAction(handler)
```

TextField Demo Program

- This progam allows a user to enter an integer in one text field, and it displays the square in a second (uneditable) text field
- The user interface uses a couple of labels to identify the purpose of the text fields to the user

TextField Demo Program

Create the two labels and the two text fields

```
// Create labels for the user interface.
Label inputLabel = new Label("Number: ");
Label outputLabel = new Label("Square of Number: ");

// Create the text fields for the user interface.
TextField inputTextField = new TextField();
TextField outputTextField = new TextField();
inputTextField.setPrefColumnCount(4);
outputTextField.setPrefColumnCount(4);
outputTextField.setEditable(false);
```

TextField Demo Program

Create an Hbox to hold the labels and text fields

```
// Create HBox and add  the labels and textfields.
HBox hBox = new HBox(10);
hBox.setAlignment(Pos.CENTER);
hBox.setPadding(new Insets(10));
hBox.getChildren().addAll(inputLabel, inputTextField,
                          outputLabel, outputTextField);
```

TextField Demo Program

Create the event handler and set it on the input text field:

```
EventHandler<ActionEvent> handler = event ->
{
   // Get Number from input text field.
   String inputText = inputTextField.getText().trim();
   int number = Integer.parseInt(inputText);

   // Write the square to the output text field.
   int square = number*number;
                outputTextField.setText(String.valueOf(squa
};
// Set the handler on the input text field
inputTextField.setOnAction(handler);
```