Event-Driven Programming with GUIs

- Slides derived (or copied) from slides created by Rick Mercer for CSc 335
Event Driven GUIs

- A Graphical User Interface (GUI) presents a graphical view of an application to users

- To build an event-driven GUI application, you must:
  - Have a well-tested model that is independent of the view
  - Make graphical components visible to the user
  - Ensure the correct things happen for each event
    - user clicks a button, or moves the mouse, or presses the enter key, ...

- Let's first consider some of Java's GUI components:
  - Pane, Button, Label, TextField
Graphical Components in JavaFX:

- JavaFX has many graphical components

Stage: window with title, border, menu, buttons

BorderPane: where we can add Buttons, Labels, … (inside the Scene)

Button: A component that can "clicked"

Label: A display area for a small amount of text

TextField: Allows editing of a single line of text
Get the app to show itself

// Show an empty stage with no components in it
public class FirstApp extends Application {

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

    @Override
    public void start(Stage stage) throws Exception {
        stage.setTitle("Our First GUI");
        BorderPane window = new BorderPane();
        Scene scene = new Scene(window, 300, 90); // 300 pixels wide, 90 tall
        stage.setScene(scene);
        // Don't forget to show the running app:
        stage.show();
    }
}
Add some components:

- So far we have an empty stage

- Let us add a Button, a Label, and a one line Editor (TextField)

- First construct three graphical components

```java
// Three different UI controls as instance variables
private Button button = new Button("Nobody is listening to me");
private Label label = new Label("Button above, text field below");
private TextField textField = new TextField("You can edit this text");
```

- Need to add these objects to the BorderPane referenced by window
Components are nodes in a graph:

- Add three components to the BorderPane as Node objects

```java
window.setTop(button);
window.setCenter(label);
window.setBottom(textField);
```

- In addition to the 3 message above, we can
  - setLeft(Node)
  - setRight(Node)
- The Node objects are in a Pane object
  - These nodes are children of the Pane
- The Pane is in a Scene object
- The Scene is in the Stage object
The 5 areas of BorderPane:

• By default, `BorderPane` objects have only five places where you can add components
  • a 2nd add wipes out the 1st

```java
window.setTop(new Button("Top"));
window.setLeft(new Button("Left"));
window.setCenter(new Button("Center"));
window.setRight(new Button("Right"));
window.setBottom(new Button("Bottom"));
```

• BTW: There is no padding or locating Nodes here
  • The layout looks odd
There are many Panes with layout strategies:

<table>
<thead>
<tr>
<th>Pane Class</th>
<th>Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>BorderPane</td>
<td>Areas for top, bottom, left, right, center</td>
</tr>
<tr>
<td>HBox, VBox</td>
<td>Lines up children horizontally or vertically</td>
</tr>
<tr>
<td>GridPane</td>
<td>Layout children in a table like grid</td>
</tr>
<tr>
<td>TilePane</td>
<td>Layout children in a grid, all the same size</td>
</tr>
<tr>
<td>FlowPane</td>
<td>Layout children left to right, top to bottom</td>
</tr>
<tr>
<td>AnchorPane</td>
<td>Children are positioned in relative position to the Pane's boundary</td>
</tr>
<tr>
<td>StackPane</td>
<td>Wraps children inside others, used to decorate such as putting a button over a colored rectangle</td>
</tr>
</tbody>
</table>
AnchorPane:

- You can change the layout strategy with a different class of Pane
- With AnchorPane, we can position children
  - specific number of pixels down from top of Pane
  - specific number of pixels from the right of the Pane

```java
AnchorPane window = new AnchorPane();
AnchorPane.setTopAnchor(button, 5.0);
AnchorPane.setRightAnchor(button, 60.0);
AnchorPane.setTopAnchor(label, 35.0);
AnchorPane.setRightAnchor(label, 60.0);
textField.setPrefWidth(280);
AnchorPane.setRightAnchor(textField, 10.0);
AnchorPane.setTopAnchor(textField, 55.0);
```
Event-Driven programming:

1. START
2. Initialization
3. Wait for input events
4. Decode events, update internal state
5. Execute requested actions
6. User requested quit?
   - No: Go back to step 3
   - Yes: STOP
Event-Driven programming:

1. User interacts with page
2. An "event" occurs
3. A piece of code runs in response
4. The page's appearance is updated/modified in some way as a result

```javascript
function myEvent() {
    ...
}
```
Event-Driven programming?

- A style of coding where a program's overall flow of execution is dictated by events
- The program loads
- The program waits for the user to generate input
- Each event causes some particular code to respond
- Need an event handler
- The overall flow of what code is determined by the user generating a series of events
What is Event-Driven programming?

- Contrast with application- or algorithm-driven control where program expects input data in a pre-determined order and timing

- Event-driven is a different programming paradigm
  - Procedural (C)
  - Object-Oriented (Java, Python)
  - Event-driven (Java, Javascript)
  - Declarative (SQL in 337 and 460)
  - Functional (ML in 372)
  - Logic (Prolog in 372)
There are many kinds of Events

- Different events that can occur in an event-driven program with a GUI
  - Mouse move/drag/click, mouse button press/release
  - Keyboard: key press/release
  - Touchscreen finger tap/drag
  - Joystick, drawing tablet, other device inputs
  - Window resize/minimize/restore/close
  - Network activity or file I/O (start, done, error)
  - Timer interrupt
  - Move a scroll bar
  - Chose a menu selection
  - Media finishes
Java’s Event Model

• Java and the operating system work together to detect user interaction
  • **Button** objects are notified when clicked
    • Send a `handle(ActionEvent)` message to registered `ActionEvent` handlers
  • **TextField** objects are notified when the user presses Enter
    • A `handle(ActionEvent)` message is sent to registered event handlers
  • When the mouse is clicked, the node under the curser is notified
    • Send a `handle(MouseEvent)` message to registered Mouse event handlers
  • When a key is pressed
    • Send a `handle(KeyEvent)` message to registered `KeyEvent` handlers
Example: ActionEvent

- The button and textField do not yet perform any action
- Let’s make something happen when
  - The button is clicked
  - The user presses enters into the textField
How to Handle Events

• Add a private inner class that will handle the event that the component generates
  • This class must implement an interface to guarantee that it has the expected method such as
    
    ```java
    public void handle(ActionEvent ae)
    ```

• Register the event handler so the component can later send the correct message to that event handler
  • Events occur anytime in the future--the event handler is waiting for user generated events such as clicking button
  • Send this message to the GUI component:
    
    ```java
    button.setOnAction(handler)
    ```
Inner Classes:

- An *inner class* is a class defined within another class.
- Inner class methods can access the data from the scope in which they are defined.
- Inner classes can be hidden from other classes in the same package.
Event 1: Add Event to Handle a button press

- Must add a class that implements `EventHandler<ActionEvent>`.

```java
EventHandler<ActionEvent> handler = new ButtonHandler();
button.setOnAction(handler);
stage.show();
}

private class ButtonHandler implements EventHandler<ActionEvent> {
    private int timesClicked;
    public ButtonHandler() {
        timesClicked = 0;
    }

    @Override
    public void handle(ActionEvent arg0) {
        button.setText("I now have a handler");
        timesClicked++;
        System.out.println("The button was clicked "+ timesClicked + " times");
        button.setText("I now have a handler");
    }
}
```
Run this program

```java
EventHandler<ActionEvent> handler = new ButtonHandler();
button.setOnAction(handler);
stage.show();
}

private class ButtonHandler implements EventHandler<ActionEvent> {
    private int timesClicked = 0;

    @Override
    public void handle(ActionEvent arg0) {
        System.out.println(arg0.toString());
        timesClicked++;
        System.out.println("The button was clicked " + timesClicked + " times");
        button.setText("I now have a handler");
    }
}
```

Our First GUI
I now have a handler
Button left, text field below
You can edit this text

```
javafx.event.ActionEvent[source=Button@23a49aa3[styleClass=button]'Nobody is listening to me']
The button was clicked 1 times
javafx.event.ActionEvent[source=Button@23a49aa3[styleClass=button]'I now have a handler']
The button was clicked 2 times
javafx.event.ActionEvent[source=Button@23a49aa3[styleClass=button]'I now have a handler']
The button was clicked 3 times
```
Event 2: Handle TextField Event:

- Must add another class that implements `EventHandler<ActionEvent>`.

```java
EventHandler<ActionEvent> handler2 = new TextFieldHandler();
textField.setOnAction(handler2);
stage.show();
}

private class TextFieldHandler implements EventHandler<ActionEvent> {
    private int enterPressed = 1;

    @Override
    public void handle(ActionEvent arg0) {
        enterPressed++;
        String text = textField.getText();
        if (enterPressed % 2 == 0) {
            textField.setText(text.toUpperCase());
        } else {
            textField.setText(text.toLowerCase());
        }
    }
}
```
Run this Program:

```java
EventHandler<ActionEvent> handler2 = new TextFieldHandler();
textField.setOnAction(handler2);
stage.show();
}

private class TextFieldHandler implements EventHandler<ActionEvent> {
    private int enterPressed = 1;
    @Override
    public void handle(ActionEvent arg0) {
        enterPressed++;
        String text = textField.getText();
        if (enterPressed % 2 == 0)
            textField.setText(text.toUpperCase());
        else
            textField.setText(text.toLowerCase());
    }
}
```