Event Driven Programming

Josh Wilcox (jw14g24)

March 25, 2025

Table of Contents

Event Driven Programming

Three different ways a program can run:

- Sequential Execution
  - Each statement in the program is executed one after the other
  - Follows a predefined path from start to finish
- Concurrent Execution
  - Multiple statements can be ran at the same time
  - Uses threads or processes to perform tasks in parallel
- Event Driven Execution
  - An event happening directly courses a sequence of events to happen
  - Program waits for specific events rather than following a predetermined sequence

Examples of where EDP is used

- GUIs
  - Clicks, Typing, Mouse Movement, etc trigger actions in the application
  - Button presses can trigger specific callback functions
- Web Servers
  - Incoming requests trigger actions and responses
  - Connection events trigger specific handlers based on URL patterns
- Gaming
  - Player actions, collisions, physics changes trigger actions
  - Input from controllers or keyboard generates game state changes
- Stocks and Shares
  - Price changes trigger automated buying or selling actions
- IoT
  - Sensor readings trigger responses in connected devices

EDP Component Structure

- Stage
  - Top level container representing the application window
  - JavaFX provides a primary stage automatically at application launch
  - We can create additional stages for dialogs, pop-ups, or multiple windows
  - Controls window properties like size, position, title, and modality
  - *Equivalent to a thread*
- Scene
  - Holds the contents of a window (the visual elements)
  - We can change the scene of a stage at any point for different views
  - Held as a *scene graph* - a hierarchical tree of nodes
  - Only one scene can be active in a stage at a time
- Nodes
  - The visual elements that make up the UI (containers and components)
  - Two main types:
    - **Parent nodes:** Can have children (e.g., `VBox`, `HBox`, `BorderPane`)
    - **Leaf nodes:** Cannot have children (e.g., `Button`, `Label`, `TextField`)
  - All nodes can receive and process events
  - Nodes form a parent-child relationship in the scene graph

Scene Graphs

- Each scene has a graph structure
- Contains exactly one **root node**
- Nodes can have children (meaning they are a container)
- Nodes can be leaves (meaning they are a component like a widget or button etc.)

Components

- JavaFX components (controls) are UI elements users interact with
- Component Creation
  - Created using constructors with various parameter options
  - Example: `Button btn = new Button("Click Me");`
  - Most controls have multiple constructor variants (empty, with label, with initial value)
- Component Properties
  - Controls have properties that affect appearance and behavior
  - Properties typically follow JavaBean naming patterns (get/set methods)
  - Many properties are exposed as observable properties (`StringProperty`, `BooleanProperty`)
  - Can be bound to other properties for automatic updates
- Common Methods
  - `setDisable(boolean)` - enable/disable interaction
  - `setVisible(boolean)` - control visibility
  - `setStyle(String)` - apply CSS styling
  - `setOnAction(EventHandler)` - attach event handlers
- Component Hierarchy
  - All JavaFX controls inherit from `Control` class
  - `Control` inherits from `Region`, which inherits from `Node`
  - This inheritance gives all controls common functionality for layout and events

Containers

- An environment to put **other components** within
- VBox
  - Positions components vertically in a single column
  - Can set spacing between elements: `new VBox(10)` creates 10px spacing
  - Control alignment with `setAlignment(Pos.CENTER)`
  - Example: `VBox vbox = new VBox(10, label, button, textField);`
- HBox
  - Positions components horizontally in a single row
  - Similar to VBox with spacing and alignment options
  - Can set margins for individual children: `HBox.setMargin(button, new Insets(10))`
  - Useful for button bars and navigation controls
- Pane
  - Simple container with no automatic layout management
  - Children must be positioned explicitly with `setLayoutX/Y`
  - Absolute positioning gives precise control: `node.setLayoutX(100); node.setLayoutY(50);`
  - Good for custom layouts or drawing applications
- StackPane
  - Places children in a z-order stack (most recent addition in front)
  - Useful for overlapping elements like backgrounds with text
  - Centers children by default
  - Good for layered UIs like cards, popups, or component overlays
  - Example: `StackPane stack = new StackPane(background, content, overlay);`
- AnchorPane
  - Anchors nodes to top, bottom, left, right, or center of the pane
  - Set anchor distances with static methods: `AnchorPane.setTopAnchor(node, 10.0)`
  - Elements can be anchored to multiple sides to control resizing behavior
  - Great for responsive layouts that resize proportionally with the window

Buttons and Event Handlers

- Buttons are interactive controls that trigger actions when clicked
- Creating a button is simple: `Button myButton = new Button("Click Me");`
- By default, buttons don't do anything when clicked
- Each button press creates an `ActionEvent` object
- Event handling process:
  - User interacts with a component (clicks a button)
  - JavaFX generates an appropriate event object
  - The event is passed to any registered event handlers
  - Handler's code executes in response to the event
- Adding an event handler to a button:

```
1 Button btn = new Button("Say Hello");
2 btn.setOnAction(e -> {
3     System.out.println("Hello World!");
4     // e is the ActionEvent object
5 });
```

- The `setOnAction()` method accepts an `EventHandler<ActionEvent>`
- Lambda expressions provide a concise way to define event handlers
- Event handlers can also be implemented as separate classes

Nested Classes

- Java allows us to define classes inside other classes - these are called nested classes
- Four types of nested classes in Java, each with distinct use cases in JavaFX applications

Static Nested Classes

- Declared with the static modifier
- Can access static members of outer class, but not instance members
- Created without an instance of the outer class
- Often used for helper classes that belong to the enclosing class

```java
// Example using a static nested class as an event handler
public class MyApplication extends Application {
    @Override
    public void start(Stage stage) {
        Button button = new Button("Click Me");
        button.setOnAction(new ButtonClickHandler());
        // ...
    }

    // Static nested class as event handler
    static class ButtonClickHandler implements EventHandler<ActionEvent> {
        @Override
        public void handle(ActionEvent event) {
            System.out.println("Button clicked!");
        }
    }
}
```

Non-static Inner Classes

3. Nested Classes
   - Static Nested Classes
   - Non-static Inner Classes
   - Local Inner Classes
   - Lambda Expressions

- Have access to all members of the enclosing class, including private members
- Always associated with an instance of the enclosing class
- Useful for implementing helper classes that need access to the outer class state
- Commonly used in JavaFX when event handlers need to access component state

```java
// Example using a non-static inner class as event handler
public class MyApplication extends Application {
    private int clickCount = 0;
    private Label statusLabel = new Label("No clicks yet");

    @Override
    public void start(Stage stage) {
        Button button = new Button("Click Me");
        button.setOnAction(new ButtonClickHandler());
        // ...
    }

    // Non-static inner class (can access instance variables)
    class ButtonClickHandler implements EventHandler<ActionEvent> {
        @Override
        public void handle(ActionEvent event) {
            clickCount++; // Can access outer class fields
            statusLabel.setText("Button clicked " + clickCount + " times");
        }
    }
}
```

Local Inner Classes

3. Nested Classes
   - Static Nested Classes
   - Non-static Inner Classes
   - Local Inner Classes
   - Lambda Expressions

- Defined inside a method or block
- Only visible within that method or block
- Can access both local variables (if final/effectively final) and class members
- Good for small specialized implementations used only in one method
- Useful when an event handler is only needed in a specific context

```
public void start(Stage stage) {
    Button button = new Button("Click Me");
    TextField inputField = new TextField();

    // Local inner class defined within method
    class ButtonClickHandler implements EventHandler<ActionEvent> {
        @Override
        public void handle(ActionEvent event) {
            // Can access local variables if effectively final
            String input = inputField.getText();
            System.out.println("Processing input: " + input);
        }
    }

    button.setOnAction(new ButtonClickHandler());
    // ...
}
```

Lambda Expressions

- Compact way to implement single-method interfaces (functional interfaces)
- Provide more concise syntax than anonymous classes
- Ideal for event handlers and simple callbacks
- Most common approach for JavaFX event handling in modern code
- Can implicitly access effectively final local variables and instance fields

```
// Example using lambda expressions for JavaFX event handling
public void start(Stage stage) {
    Button button = new Button("Click Me");
    TextField textField = new TextField();
    Label resultLabel = new Label();

    // Lambda expression for button event handling
    button.setOnAction(event -> {
        String input = textField.getText();
        resultLabel.setText("You entered: " + input);
    });

    // Lambda for property change listener
    textField.textProperty().addListener((obs, oldText, newText) ->
        System.out.println("Text changed from: " + oldText + " to: " + newText));

    // Lambda for mouse events
    button.setOnMouseEntered(e -> button.setStyle("-fx-background-color: lightblue;"));
    button.setOnMouseExited(e -> button.setStyle(""));
}
```