14. Interfacce Utente Grafiche (GUI) con JavaFX

Questo capitolo introduce **JavaFX**, una libreria moderna e potente per la creazione di interfacce utente grafiche (GUI) in Java. Verranno esplorati i suoi concetti fondamentali, l'architettura, la gestione degli eventi, i layout e l'integrazione con FXML e CSS per una progettazione più modulare.

8.1. Introduzione a JavaFX

JavaFX è una libreria Java per la creazione di "Rich Applications" multi-piattaforma. È stata progettata per sostituire gradualmente Swing e offre un approccio più moderno e flessibile allo sviluppo di GUI.

- *Provenienza*: 18-gui_slides.pdf, Pagina 4

8.1.1. Storia e Evoluzione di JavaFX

- **2008 (v. 1.0 – 2.2)**: Disponibile come libreria stand-alone.
- **Java 8 (v. JavaFX 8)**: Introdotta "stabilmente" nel JDK con l'idea di sostituire Swing.
- **Java 11 in poi**: Torna ad essere una libreria stand-alone, parte del progetto OpenJDK e open-source (https://openjfx.io). Questo significa che, a differenza di Swing che è inclusa nel JDK, per utilizzare JavaFX in progetti moderni è spesso necessario aggiungerla come dipendenza esterna (es. tramite Maven o Gradle).
- **Requisiti**: Nel 2022, JavaFX 19 richiede JDK >= 11.
- *Provenienza*: 18-gui_slides.pdf, Pagina 4-5

8.1.2. Funzionalità Principali di JavaFX

JavaFX offre un set di funzionalità avanzate per la creazione di GUI moderne e di alta qualità:

- **Java APIs**: La libreria è interamente scritta in Java, fornendo un set ricco di classi e interfacce.
- **FXML**: Un linguaggio dichiarativo basato su XML per definire la struttura della GUI. Permette di separare il design dell'interfaccia dalla logica applicativa.
- **CSS**: Un linguaggio flessibile per specificare lo stile degli elementi della GUI, simile al CSS utilizzato per le pagine web.
- **MVC-friendly**: Supporta nativamente pattern di progettazione come MVC (Model-View-Controller) e le sue varianti, grazie a FXML, proprietà osservabili e data binding.
- **Graphics API**: Supporto nativo per la grafica 2D e 3D (geometrie, camere, luci), e la possibilità di disegnare direttamente su una superficie (canvas).
- **Supporto Multi-touch e Hi-DPI**: Gestisce funzionalità multi-touch (es. `SwipeEvent`) e garantisce una buona visualizzazione su schermi ad alta densità di pixel.
- **Interoperabilità con Swing**: Permette l'integrazione bidirezionale con GUI Swing esistenti (tramite `JFXPanel` per includere componenti JavaFX in Swing, e `SwingNode` per includere componenti Swing in JavaFX).
- *Provenienza*: 18-gui_slides.pdf, Pagina 5-6

8.2. Astrazioni Fondamentali di un'Applicazione JavaFX

Un'applicazione JavaFX è costruita su una gerarchia di astrazioni chiave che definiscono la sua struttura visiva e il suo ciclo di vita.

8.2.1. `Application`

- La classe principale di un'applicazione JavaFX deve estendere `javafx.application.Application`.
- Consente di definire metodi "hook" sul ciclo di vita dell'applicazione:
  - `init()`: Chiamato prima di `start()`, utile per inizializzazioni non-GUI.
  - `start(Stage primaryStage)`: L'entry point effettivo dell'applicazione JavaFX. Riceve lo `Stage` primario creato dalla piattaforma. Qui si costruisce la scena e la si associa allo stage.
  - `stop()`: Chiamato alla terminazione dell'applicazione, utile per rilasciare risorse.

- Il metodo `main()` dell'applicazione Java deve chiamare `Application.launch(App.class, args)` per avviare il runtime JavaFX. È consigliabile definire `main()` in una classe separata dalla classe `App` per evitare problemi legati al module path di JavaFX.
- *Provenienza*: 18-gui_slides.pdf, Pagina 8-9, 15

8.2.2. `Stage`

- Il `Stage` (rappresentato dalla classe `javafx.stage.Stage`) è il contenitore esterno di una GUI JavaFX, equivalente a un `JFrame` in Swing.
- Corrisponde a una finestra del sistema operativo (es. una finestra desktop).
- Ogni `Stage` può mostrare una sola `Scene` alla volta, impostabile tramite `Stage#setScene(Scene)`.
- Deve essere mostrato invocando `stage.show()`.
- *Provenienza*: 18-gui_slides.pdf, Pagina 7, 12 (diagramma), 15

8.2.3. `Scene`

- Una `Scene` (rappresentata da `javafx.scene.Scene`) è il contenuto visualizzabile su uno `Stage`.
- Contiene il cosiddetto **scene graph**, che è una gerarchia di nodi (`Node`) che definiscono l'interfaccia utente.
- Il nodo radice del scene graph è impostato tramite `Scene#setRoot(Parent)`.
- La dimensione della `Scene` può essere specificata o calcolata automaticamente in base al contenuto. Se il nodo radice è ridimensionabile (es. `Region`), il ridimensionamento della scena causerà un aggiustamento del layout.
- *Provenienza*: 18-gui_slides.pdf, Pagina 7, 12 (diagramma), 24

8.2.4. `Node` e Scene Graph

- Un `Node` è un elemento o componente della scena (es. un pulsante, un'etichetta, un pannello).
- Ogni `Node` ha sia una parte di "view" (aspetto) che una parte di "controller" (comportamento, tramite event handler).
- I nodi hanno proprietà (con supporto al binding) e possono generare eventi.
- Possono essere organizzati gerarchicamente: la sottoclasse `Parent` rappresenta nodi che possono avere figli (recuperabili con `getChildren()`).
- Ogni nodo ha un ID univoco, coordinate locali, può subire trasformazioni (es. rotazione), ha un "bounding rectangle" associato e può essere stilizzato con CSS.
- Sottoclassi importanti di `Node` includono `SwingNode`, `Canvas` e `Parent`.
- *Provenienza*: 18-gui_slides.pdf, Pagina 11, 12 (diagramma)

Struttura riassuntiva di un'applicazione JavaFX:

Elemento	Descrizione	Corrispondenza Swing
`Application`	Classe principale, gestisce il ciclo di vita.	N/A (gestione del `main`)
`Stage`	Finestra di alto livello del sistema operativo.	`JFrame`
`Scene`	Contenuto visualizzato su uno Stage, contiene il scene graph.	`JPanel` (come contenitore principale)
`Parent`	Nodo che può contenere altri nodi (layout manager).	`JPanel` (come contenitore intermedio)
`Node`	Componente UI atomico (pulsante, etichetta, campo di testo).	`JComponent`

- *Provenienza*: 18-gui_slides.pdf, Pagina 12-13 (diagrammi UML)

8.2.5. Esempio Base di un'Applicazione JavaFX

```
import javafx.application.Application;
import javafx.scene.Group; // Un tipo di Parent
import javafx.scene.Scene;
import javafx.stage.Stage;

// Classe principale dell'applicazione JavaFX
```

```
public class App extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        // 1. Creo il nodo radice della scena. Group è un contenitore semplice.
        Group root = new Group();

        // 2. Creo la scena, specificando il nodo radice e le dimensioni iniziali.
        Scene scene = new Scene(root, 500, 300);

        // 3. Imposto il titolo dello stage (finestra).
        stage.setTitle("JavaFX Demo");

        // 4. Associo la scena allo stage.
        stage.setScene(scene);

        // 5. Mostro lo stage.
        stage.show();
    }
}

// Classe separata per il metodo main (runner)
class Main {
    public static void main(String[] args) {
        // Avvia l'applicazione JavaFX
        Application.launch(App.class, args);
    }
}
```

- *Provenienza*: 18-gui_slides.pdf, Pagina 8-9

8.3. Proprietà e Data Binding

Una delle caratteristiche più potenti di JavaFX è il sistema di **proprietà osservabili** e il **data binding**, che facilita la sincronizzazione dei dati tra il modello e la vista.

8.3.1. Proprietà Osservabili (`Property<T>`)

- Ciascun nodo (componente) in JavaFX espone diverse proprietà osservabili (istanze di `javafx.beans.property.Property<T>`).
- Queste proprietà possono riguardare l'aspetto (es. `sizeProperty()`, `positionProperty()`), il contenuto (es. `textProperty()`, `valueProperty()`) o il comportamento (es. `eventHandlerProperty()`).
- Una `Property<T>` è un `ObservableValue<T>`, il che significa che è un valore a cui possono essere associati dei `ChangeListener`. Quando il valore della proprietà cambia, i listener registrati vengono notificati.
- Ogni proprietà JavaFX `xxx` di tipo `T` ha (opzionalmente) getter/setter `getXxx()` e `setXxx()`, e un metodo `xxxProperty()` che restituisce l'oggetto `Property<T>` associato. Ad esempio, un `TextField` offre `getText()`, `setText(String)` e `textProperty():Property<String>`.
- *Provenienza*: 18-gui_slides.pdf, Pagina 16, 18

8.3.2. Data Binding

Il **data binding** è il meccanismo che consente di collegare due proprietà tra loro, in modo che una modifica a una proprietà si rifletta automaticamente nell'altra.

- **Binding Unidirezionale (`bind(ObservableValue<? extends T> observable)`)**: Una proprietà viene legata a un'altra, diventando un "listener" dei suoi cambiamenti. La proprietà legata si aggiornerà automaticamente quando la proprietà sorgente cambia.
- **Binding Bidirezionale (`bindBidirectional(Property<T> other)`)**: Due proprietà sono legate in modo che una modifica a una si rifletta nell'altra, e viceversa. Questo è particolarmente utile per sincronizzare i dati tra i componenti dell'interfaccia utente (es. un `TextField` e una `Label`).

- `unbind()` / `unbindBidirectional()`: Metodi per scollegare le proprietà.

Esempio di Binding Bidirezionale:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class BindingExample extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        final TextField input = new TextField();
        final Label mirror = new Label();

        // Connette la label con il valore del textfield in modo bidirezionale
        mirror.textProperty().bindBidirectional(input.textProperty());
        mirror.setText("default"); // Il valore iniziale della label si propaga al textfield

        HBox root = new HBox(10); // HBox con 10px di spaziatura
        root.getChildren().addAll(new Label("Input:"), input, new Label("Mirror:"), mirror);

        stage.setTitle("JavaFX - Binding Example");
        stage.setScene(new Scene(root, 400, 100));
        stage.show();
    }
}
```

- *Provenienza*: 18-gui_slides.pdf, Pagina 18

8.4. Layout in JavaFX

JavaFX fornisce un ricco set di "layout pane" (sottoclassi di `Parent` e `Region`) che regolano il posizionamento e il dimensionamento dei nodi figli.

8.4.1. Gerarchia dei Layout Pane

La gerarchia dei layout in JavaFX è ben strutturata:

- `Node` **(abstract)**: Classe base per tutti i componenti.
  - `Parent` **(abstract)**: Nodi che possono avere figli (es. contenitori, layout manager).
    - `Group` **(non-resizable)**: Gestisce un insieme di figli posizionati in coordinate fisse. Qualsiasi trasformazione o effetto applicato al `Group` viene applicato a tutti i suoi figli.
    - `Region` **(resizable)**: Classe base per tutti i controlli UI e i layout ridimensionabili.
      - `Pane`: Classe base per la maggior parte dei layout general-purpose.
      - **Controlli UI specifici**: `TabPane`, `TitledPane`, `SplitPane`, `Accordion`, `ToolBar`.
      - **Layout Pane Specifici**:
        - `StackPane`: Posiziona i figli uno sopra l'altro, centrati.
        - `HBox` / `VBox`: Organizza i figli in una singola riga orizzontale (`HBox`) o verticale (`VBox`).
        - `TilePane`: Organizza i figli in una griglia di "tessere" (tiles) della stessa dimensione.
        - `FlowPane`: Posiziona i figli in base al loro flusso, andando a capo quando non c'è più spazio (simile a `FlowLayout` di Swing).
        - `AnchorPane`: Permette di "ancorare" i figli a bordi specifici del contenitore.

    - `BorderPane`: Divide l'area in cinque regioni (TOP, BOTTOM, LEFT, RIGHT, CENTER), simile a `BorderLayout` di Swing.
    - `GridPane`: Organizza i figli in una griglia di righe e colonne.
- *Provenienza*: 18-gui_slides.pdf, Pagina 19-20 (diagramma UML)

8.4.2. Aggiungere Componenti ai Layout

Il metodo `ObservableList<Node> getChildren()` di qualsiasi nodo/layout restituisce la lista dei nodi figli. I componenti possono essere aggiunti (`add(Node)`) o aggiunti in blocco (`addAll(Node...)`).

Esempi di utilizzo dei Layout Pane:

- **`HBox` / `VBox`**:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.HBox; // o VBox
import javafx.stage.Stage;

public class HBoxExample extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        final Label lbl = new Label("Label text here...");
        final Button btn = new Button("Click me");

        // HBox organizza i figli orizzontalmente
        final HBox root = new HBox(10); // 10px di spaziatura tra i figli
        root.getChildren().add(btn);
        root.getChildren().add(lbl);

        stage.setTitle("JavaFX - Example HBox");
        stage.setScene(new Scene(root, 300, 250));
        stage.show();
    }
}
```

    - *Provenienza*: 18-gui_slides.pdf, Pagina 17 (adattato)
- **`BorderPane`**:

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class BorderPaneExample extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        Button top = new Button("Top");
        Button bottom = new Button("Bottom");
        Button left = new Button("Left");
        Button right = new Button("Right");
        Label center = new Label("Center Content");
```

```
        BorderPane root = new BorderPane();
        root.setTop(top);
        root.setBottom(bottom);
        root.setLeft(left);
        root.setRight(right);
        root.setCenter(center);

        // Allineamento dei componenti nelle regioni (es. in alto al centro)
        BorderPane.setAlignment(top, Pos.CENTER);

        stage.setTitle("JavaFX - BorderPane Example");
        stage.setScene(new Scene(root, 400, 300));
        stage.show();
    }
}
```

- *Provenienza*: 18-gui_slides.pdf, Pagina 22 (concetti di base)

- `GridPane`:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;
import java.time.Month;

public class GridPaneExample extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        GridPane gp = new GridPane();
        gp.setGridLinesVisible(true); // Utile per il debug

        // Aggiungo etichette per ogni mese
        for (Month m : Month.values()) {
            Label l = new Label(m.name());
            // Imposto le costrizioni di griglia per la label
            // colonna = (valore_mese - 1) / 4, riga = (valore_mese - 1) % 4
            int columnIndex = (m.getValue() - 1) / 4;
            int rowIndex = (m.getValue() - 1) % 4;
            GridPane.setConstraints(l, columnIndex, rowIndex);
            gp.getChildren().add(l); // Aggiungo la label al GridPane
        }

        stage.setTitle("JavaFX - GridPane Example");
        stage.setScene(new Scene(gp, 500, 300));
        stage.show();
    }
}
```

- *Provenienza*: 18-gui_slides.pdf, Pagina 22 (concetti di base)

8.5. Gestione degli Eventi in JavaFX

Come Swing, JavaFX utilizza un modello basato su eventi per l'interazione utente.

8.5.1. Concetto di Evento e `EventHandler`

- Gli eventi (`javafx.event.Event`) possono essere generati dall'interazione dell'utente con gli elementi grafici (es. click del mouse, pressione di un tasto).
- Ogni evento ha una sorgente (`event source`), un target (`event target`) e un tipo (`event type`).
- Gli eventi possono essere "consumati" (`consume()`) per impedire che vengano propagati ulteriormente.
- Gli eventi sono gestiti tramite **event handlers**, che sono oggetti che implementano l'interfaccia funzionale `EventHandler<T extends Event>` e il suo metodo `void handle(T event)`.
- Ogni nodo può registrare uno o più event handler, tipicamente tramite metodi `setOn...()` (es. `setOnMouseClicked()`, `setOnAction()`).
- *Provenienza*: 18-gui_slides.pdf, Pagina 25

8.5.2. Processamento degli Eventi

Il processo di gestione degli eventi in JavaFX segue una "event route" e si articola in fasi:

1. **Selezione dell'Event Target**: Il nodo su cui si è verificato l'evento (es. il pulsante cliccato).
2. **Costruzione dell'Event Route**: Tipicamente dallo `Stage` fino all'event target.
3. **Percorrimento dell'Event Route**:
    - **Capture Phase**: Gli `event filter` vengono eseguiti dalla testa (Stage) alla coda (event target) della route. I filtri possono intercettare e consumare l'evento prima che raggiunga il target.
    - **Event Bubbling (o Target Phase + Bubbling Phase)**: Gli `event handler` vengono eseguiti dalla coda (event target) alla testa (Stage) della route.
- *Provenienza*: 18-gui_slides.pdf, Pagina 25

8.5.3. Esempio di Gestione Eventi con Lambda

Le lambda expressions (introdotte nel Capitolo 5) sono il modo preferito e più conciso per implementare gli `EventHandler` in JavaFX, dato che `EventHandler` è un'interfaccia funzionale.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.input.MouseEvent; // Per MouseEvent
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class EventHandlingExample extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        final Label lbl = new Label("Label text here...");
        final Button btn = new Button("Click me");

        // Gestione del click sul pulsante usando una lambda
        // setOnMouseClicked è un metodo convenzionale per registrare un EventHandler<MouseEvent>
        btn.setOnMouseClicked(event -> {
            lbl.setText("Hello, JavaFX World!");
            System.out.println("Pulsante cliccato alle: " + event.getSceneX() + ", " + event.getSceneY());
        });

        // Alternativa più generica con addEventHandler
        // btn.addEventHandler(MouseEvent.MOUSE_CLICKED, e -> lbl.setText("Hello, JavaFX World!"));

        final HBox root = new HBox(10);
        root.getChildren().addAll(btn, lbl);

        stage.setTitle("JavaFX - Event Example");
        stage.setScene(new Scene(root, 300, 250));
```

```
        stage.show();
    }
}
```

- *Provenienza*: 18-gui_slides.pdf, Pagina 26

8.5.4. JavaFX Application Thread (JFXAT)

Similmente a Swing con l'EDT, JavaFX ha un singolo thread dedicato alla gestione degli eventi e all'aggiornamento della GUI: il **JavaFX Application Thread (JFXAT)**.

- Tutte le modifiche allo scene graph (ovvero, agli elementi visibili della GUI) devono essere effettuate sul JFXAT.
- Se un'operazione lunga o bloccante viene eseguita sul JFXAT, l'interfaccia utente diventerà non responsiva.
- Per eseguire operazioni lunghe in background, si devono usare thread separati e poi utilizzare `Platform.runLater(Runnable)` per accodare le modifiche alla GUI sulla coda degli eventi del JFXAT. Questo è l'analogo di `SwingUtilities.invokeLater()` in Swing.
- *Provenienza*: 18-gui_slides.pdf, Pagina 29

8.6. FXML: Separazione del Design dalla Logica

FXML è un linguaggio di markup basato su XML che consente di separare la definizione della struttura della GUI dal codice Java che ne gestisce il comportamento.

8.6.1. Motivazioni e Vantaggi di FXML

- **Separazione dei Ruoli**: Permette ai designer UX di lavorare sul layout della GUI (in FXML) e agli sviluppatori di concentrarsi sulla logica applicativa (in Java), facilitando la collaborazione.
- **Dichiaratività**: Descrive la GUI in modo dichiarativo, rendendo il codice più leggibile e manutenibile rispetto alla creazione programmatica di GUI complesse.
- **Facilità di Modifica**: Modificare il layout o lo stile della GUI è più semplice modificando il file FXML/CSS che ricompilando il codice Java.
- *Provenienza*: 18-gui_slides.pdf, Pagina 31-32

8.6.2. Struttura di un File FXML

- Un file FXML (con estensione `.fxml`) è un file XML valido.
- Inizia con il tag `<?xml version="1.0" encoding="UTF-8"?>`.
- Il tag radice del documento XML corrisponde al nodo radice del scene graph (es. `<VBox>`).
- Gli attributi `xmlns` e `xmlns:fx` sono obbligatori nel tag radice. Il namespace `fx` raccoglie nodi relativi al processamento interno del descrittore FXML.
- I componenti sono specificati tramite tag specifici (es. `<Button>`, `<Label>`).
- Le proprietà dei componenti sono specificate come attributi (es. `text="Say Hello!"`) o come tag annidati.
- I nodi figli sono specificati all'interno del tag `<children>`.
- L'attributo `fx:id` permette di assegnare un ID univoco a un nodo, che può essere poi referenziato dal codice Java.
- Il tag `<?import ... ?>` è equivalente all'import di Java e specifica i package in cui recuperare le classi dei componenti.
- *Provenienza*: 18-gui_slides.pdf, Pagina 33-35

Esempio di GUI in FXML:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<VBox xmlns="http://javafx.com/javafx"
      xmlns:fx="http://javafx.com/fxml">
   <children>
```

```
        <Button fx:id="btn"
                alignment="CENTER"
                text="Say Hello!"
                textAlignment="CENTER" />
        <Label fx:id="lbl"
               alignment="CENTER_LEFT"
               text="Label Text Here!"
               textAlignment="LEFT" />
    </children>
</VBox>
```

- *Provenienza*: 18-gui_slides.pdf, Pagina 34

8.6.3. Caricare FXML con `FXMLLoader`

Per collegare il design della GUI descritto in FXML al codice Java, si utilizza la classe `javafx.fxml.FXMLLoader`.

```
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Example3 extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        // Carica il file FXML dal classpath
        // Si suppone che layouts/main.fxml sia nel classpath
        Parent root = FXMLLoader.load(ClassLoader.getSystemResource("layouts/main.fxml"));
        Scene scene = new Scene(root, 500, 250); // Crea la scena con il nodo radice caricato
        stage.setTitle("JavaFX - Example 3");
        stage.setScene(scene);
        stage.show();
    }

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

- *Provenienza*: 18-gui_slides.pdf, Pagina 36-37

8.6.4. Controller della GUI e Node Injection (`@FXML`)

Per implementare correttamente il pattern MVC in JavaFX, è opportuno specificare un oggetto controller per ciascuna GUI definita in FXML.

- **Associazione Controller**: Il nodo radice della GUI nel file FXML deve definire l'attributo `fx:controller` con il nome pienamente qualificato della classe che fungerà da controller.

  ```
  <VBox fx:controller="it.unibo.oop.lab.javafx.UIController">
    <!-- ... -->
  </VBox>
  ```

- **Iniezione dei Nodi (`@FXML`)**: Nella classe controller, l'annotazione `@FXML` viene utilizzata per iniettare automaticamente i riferimenti ai nodi della GUI. I nomi delle variabili d'istanza annotate devono corrispondere agli `fx:id` dei nodi nel file FXML.

- **Associazione Event Handler**: I metodi che gestiscono gli eventi possono essere associati direttamente nel file FXML usando la sintassi `on<EventType>="#methodName"`.

Esempio Completo (Controller):

```
import javafx.fxml.FXML;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.input.MouseEvent; // Importa MouseEvent se usi onMouseClicked

public class UIController {
    @FXML // Inietta il riferimento alla Label con fx:id="lbl"
    private Label lbl;
    @FXML // Inietta il riferimento al Button con fx:id="btn"
    private Button btn;

    // Metodo che gestisce l'evento di click del pulsante (associato in FXML con onMouseClicked="#btnOnClickHandle
r")
    @FXML
    public void btnOnClickHandler(MouseEvent event) { // Il parametro MouseEvent è facoltativo se non usato
        lbl.setText("Hello, World!");
        System.out.println("Pulsante cliccato dal controller!");
    }
}
```

- *Provenienza*: 18-gui_slides.pdf, Pagina 38-42

8.6.5. Scene Builder

Scene Builder è uno strumento visuale (un GUI Builder) che permette di creare GUI JavaFX in modalità drag-and-drop. Consente di progettare l'interfaccia trascinando i componenti e configurandone le proprietà, e poi esporta il design come file FXML. È uno strumento esterno al JDK, sviluppato da Gluon.

- *Provenienza*: 18-gui_slides.pdf, Pagina 43-44

8.7. Stile con CSS

JavaFX supporta l'applicazione di stili ai componenti della GUI utilizzando fogli di stile CSS, in modo simile allo sviluppo web.

8.7.1. Applicazione dello Stile CSS

- **File CSS Esterni**: Si possono definire stili in file `.css` esterni e caricarli nella `Scene` programmaticamente:

  ```
  Scene scene = new Scene(pane);
  scene.getStylesheets().add(ClassLoader.getSystemResource("css/scene.css").toExternalForm());
  ```

- **Stile Inline**: Si possono applicare stili direttamente ai nodi usando il metodo `setStyle()`:

  ```
  HBox buttons = new HBox();
  buttons.setStyle("-fx-border-color: red;");
  ```

- **Classi di Stile CSS**: Si possono assegnare classi di stile ai nodi (`getStyleClass().add("my-class")`) e poi definirne lo stile nel CSS.
- **Nel File FXML**: Si può collegare un foglio di stile direttamente nel file FXML al nodo radice:

  ```
  <GridPane id="pane" stylesheets="css/scene.css"> ... </GridPane>
  ```

- *Provenienza*: 18-gui_slides.pdf, Pagina 45

8.7.2. Selettori e Proprietà CSS in JavaFX

JavaFX ha le proprie convenzioni per i selettori e le proprietà CSS:

- **Selettori di Tipo**: Corrispondono ai nomi delle classi dei componenti (es. `.label` per tutte le `Label`).
- **Selettori ID**: Usano l'attributo `fx:id` preceduto da `#` (es. `#myButton`).
- **Proprietà**: Le proprietà CSS di JavaFX sono prefissate con `fx-` (es. `fx-font-size`, `fx-padding`, `fx-background-color`).

Esempio di file CSS (`scene.css`):

```
/* Stile per un nodo specifico con ID "myButton" */
#myButton {
    -fx-padding: 0.5em; /* Padding interno */
}

/* Stile per tutte le Label */
.label {
    -fx-font-size: 30pt; /* Dimensione del font */
    -fx-text-fill: blue; /* Colore del testo */
}

/* Stile per una classe CSS personalizzata */
.buttonrow {
    -fx-border-color: green;
    -fx-border-width: 2px;
}
```

- *Provenienza*: 18-gui_slides.pdf, Pagina 45

8.8. Organizzazione delle Applicazioni Grafiche con MVC (JavaFX)

Anche con JavaFX, il pattern **Model-View-Controller (MVC)** rimane un approccio strutturato e altamente raccomandato per la progettazione di applicazioni GUI complesse. La sua applicazione in JavaFX è facilitata dalle proprietà osservabili e da FXML.

8.8.1. MVC in JavaFX: Un Esempio Dettagliato ("DrawNumber")

Il PDF "18-gui_slides.pdf" riprende l'esempio "DrawNumber" (già visto per Swing nel Capitolo 7) e lo adatta all'architettura JavaFX con MVC, mostrando come le proprietà osservabili (`Property<T>`) e il data binding (`bind()`, `bindBidirectional()`) possano semplificare la sincronizzazione tra Model e View.

- `DrawNumber` **(Model Interface)**: Definisce le operazioni di dominio (es. `reset()`, `attempt(int n)`).
  - *Provenienza*: 18-gui_slides.pdf, Pagina 52
- `DrawNumberObservable` **(Model Observable Interface)**: Estende `DrawNumber` e aggiunge proprietà osservabili (`Property<Integer>`, `Property<Optional<Integer>>`, `Property<Optional<DrawResult>>`) per esporre lo stato del modello in modo che la View possa osservarlo.
  - *Provenienza*: 18-gui_slides.pdf, Pagina 53
- `DrawNumberImpl` **(Model Implementation)**: Implementa `DrawNumberObservable`, utilizzando `SimpleObjectProperty` per le proprietà osservabili. La logica del gioco aggiorna queste proprietà, e la View si aggiorna automaticamente tramite binding.
  - *Provenienza*: 18-gui_slides.pdf, Pagina 54-56
- `DrawNumberView` **(View Interface)**: Definisce le operazioni per visualizzare informazioni e notificare il Controller (es. `setObserver()`, `start()`, `result()`, `numberIncorrect()`, `displayError()`).
  - *Provenienza*: 18-gui_slides.pdf, Pagina 57
- `DrawNumberViewObserver` **(Controller Interface/Listener)**: Interfaccia implementata dal Controller per ricevere notifiche dalla View (es. `newAttempt(int n)`, `resetGame()`, `quit()`).

    - *Provenienza*: 18-gui_slides.pdf, Pagina 57
- `DrawNumberViewImpl` **(View Implementation)**: Implementa la GUI usando componenti JavaFX. Nel costruttore, riceve il `DrawNumberObservable` (il Model) e stabilisce i binding tra le proprietà del Model e i componenti della GUI (es. una `Label` che mostra i tentativi rimanenti si lega a `model.remainingAttemptsProperty()`).
    - *Provenienza*: 18-gui_slides.pdf, Pagina 58-60
- `DrawNumberFXApplication` **(Controller Implementation)**: Estende `Application` e implementa `DrawNumberViewObserver`. Inizializza il Model e una o più View. Gestisce le interazioni tra View e Model, come nel caso Swing, ma sfrutta il JFXAT per le operazioni GUI.
    - *Provenienza*: 18-gui_slides.pdf, Pagina 61-62

Questo esempio evidenzia come JavaFX, con le sue proprietà osservabili e il data binding, possa semplificare notevolmente la sincronizzazione tra Model e View in un'architettura MVC, riducendo la quantità di codice "boilerplate" nel Controller per aggiornare la View.

8.8.2. Linee Guida per MVC in JavaFX

Le linee guida per MVC rimangono simili a quelle per Swing:

- **Progettare le 3 Interfacce**: Definire chiaramente le interfacce per Model, View e Controller.
- **Tecnologia GUI Interna alla View**: La tecnologia JavaFX dovrebbe essere contenuta all'interno dell'implementazione della View.
- **Implementazione Separata e Test**: Implementare e testare Model, View e Controller separatamente.
- **Controller Esile**: Il Controller dovrebbe coordinare Model e View, ma non contenere logica di business o di presentazione complessa.
- *Provenienza*: 18-gui_slides.pdf, Pagina 62 (concetti ripresi da 17-swing_slides.pdf, Pagina 63)

Questo conclude il Capitolo 8, che ha esplorato le GUI con JavaFX. Ho cercato di spiegare ogni concetto in modo approfondito, fornendo esempi chiari e integrando le informazioni con dettagli tratti dalle ricerche sul web per offrire una visione completa.

Sono pronto per il prossimo capitolo quando lo sarai anche tu!