Introduzione a JavaFX

Programmazione ad Oggetti – Lab10

Docenti: Roberto **Casadei**, Danilo **Pianini**
Tutor: Luca **Deluigi**

C.D.S. Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Campus di Cesena

4 aprile 2023

Outline

Outline

Event Dispatch Thread (EDT) in Swing

- E' il thread deputato alla gestione degli eventi della GUI di un'applicazione Swing-based
- Avviato dalla JVM alla creazione del primo JFrame
  - (nota) L'applicazione non termina al completamento del main
- La gestione di tutti gli eventi relativi a componenti della GUI è demandata all'EDT
  - Ciascun evento è gestito a condizione che la gestione di tutti quelli precedenti sia terminata
  - Si presuppone che la gestione di ciascun evento non implichi una situazione di *stallo* dell'applicazione (dell'EDT)
- **Swing non è thread-safe!**
  - Si presuppone che l'EDT sia l'unico thread ad accedere alla GUI
  - Si utilizza la libreria `SwingUtilities` – i metodi `invokeLater()` and `invokeAndWait()` – per accedere alla GUI da altri thread

Limiti (e svantaggi) di Swing

- Non supporta sviluppo di applicazioni moderne multi-piattaforma (desktop, web, mobile) e multi-dispositivo (ad es. quelli touch)
- Non vi è un supporto standard per GUI dichiarative
- Non supporta la specifica separata degli stili (CSS)
- Non vi è un supporto per animazioni ed effetti
- Non propone alcun supporto nativo per la gestione del 3D
- Il modello non è sempre consistente (ad es. i layout manager non sono nodi)

Outline

JavaFX

- Libreria Java per la creazione di GUI per Rich Applications multi-piattaforma
  - Disponibile dal 2008 (v. 1.0 – 2.2) come libreria stand-alone
  - Presente "*stabilmente*" nel JDK da Java 8 (v. JavaFX 8)
  - ~~Introdotto ufficialmente in Java con l'idea di sostituire (gradualmente) Swing~~
  - Torna ad essere una libreria stand-alone da Java 11: è opensource e parte del progetto OpenJDK – `https://openjfx.io`
- Propone un look-and-feel personalizzabile
  - La descrizione dello stile/aspetto dei componenti della GUI è separato dalla relativa implementazione
  - Segue il pattern MVC
- Consente la creazione di GUI moderne, di qualità e ben adattabili a qualunque piattaforma e supporto hardware

Outline

JavaFX: Key Features (1/2)

Java APIs

- Libreria che include classi e interfacce scritte in Java
- Nel 2020, la versione più recente, JavaFX 15, richiede JDK ≥ 11

FXML (e CSS per lo stile)

- FXML è un linguaggio dichiarativo per definire la GUI di un'applicazione JavaFX-based
- Il suo impiego non è indispensabile, ma fortemente consigliato per una buona *separation of concerns*

Interoperabilità bidirezionale con Swing

- GUI Swing esistenti possono includere componenti JavaFX (cf. `JFXPanel`)
- E' possibile inserire componenti Swing in interfacce JavaFX (cf. `SwingNode`)

JavaFX: Key Features (2/2)

Graphics API

- Supporto nativo per la grafica 3D (geometrie, camere, luci)
- Abilita la possibilità di disegnare direttamente sulla superficie (canvas) dell'applicazione

Supporto per schermi Multi-touch e Hi-DPI

- Fornisce il supporto per funzionalità multi-touch (cf. `SwipeEvent`), in funzione della piattaforma in cui l'applicazione è in esecuzione
- Garantisce una buona visualizzazione della GUI anche su schermi ad alta densità

Outline

Elementi fondamentali (1/4)

Stage

- Il contenitore (esterno) dove la GUI sarà visualizzata
  - es. una finestra del S.O.
  - Equivalente al `JFrame` di Swing
  - Non è compito del programmatore creare una sua istanza.
- `https://openjfx.io/javadoc/15/javafx.graphics/javafx/stage/Stage.html`

Scene

- Rappresenta il contenuto di uno Stage (una *pagina* della GUI)
  - ogni Stage può avere più istanze diverse di Scene
- Di fatto, è un container di Node(s)
- `https://openjfx.io/javadoc/15/javafx.graphics/javafx/scene/Scene.html`

Elementi fondamentali (2/4)

Application

- `Application`: entry point di un'applicazione JavaFX application
- Consente di definire metodi hook sul ciclo di vita dell'applicazione (`init`, `start`, `stop`, ...)
  `https://openjfx.io/javadoc/15/javafx.graphics/javafx/application/Application.html`

Esempio: GUI vuota

```
1  public class App extends javafx.application.Application {
2      @Override
3      public void start(Stage stage) throws Exception {
4          Group root = new Group();
5          Scene scene = new Scene(root, 500, 300);
6          stage.setTitle("JavaFX Demo");
7          stage.setScene(scene);
8          stage.show();
9      }
10 }
```

Applicazione JavaFX: runner

```
1 import javafx.application.Application;
2
3 public class Main {
4     public static void main(String[] args) {
5         // App è la classe definita nella slide precedente
6         Application.launch(App.class, args);
7     }
8 }
```

- ATTENZIONE: per motivi tecnici che non approfondiremo, definire il metodo `main()` chiamante `launch()` dentro alla classe `App` (che estende `Application`) può risultare nel seguente errore: "Error: JavaFX runtime components are missing, and are required to run this application"
- Di conseguenza, si consiglia di definire `main` in una classe separata da quella dell'applicazione JavaFX

Ciclo di vita di applicazioni JavaFX

L'avvio mediante `Application.launch(App.class)` comporta:

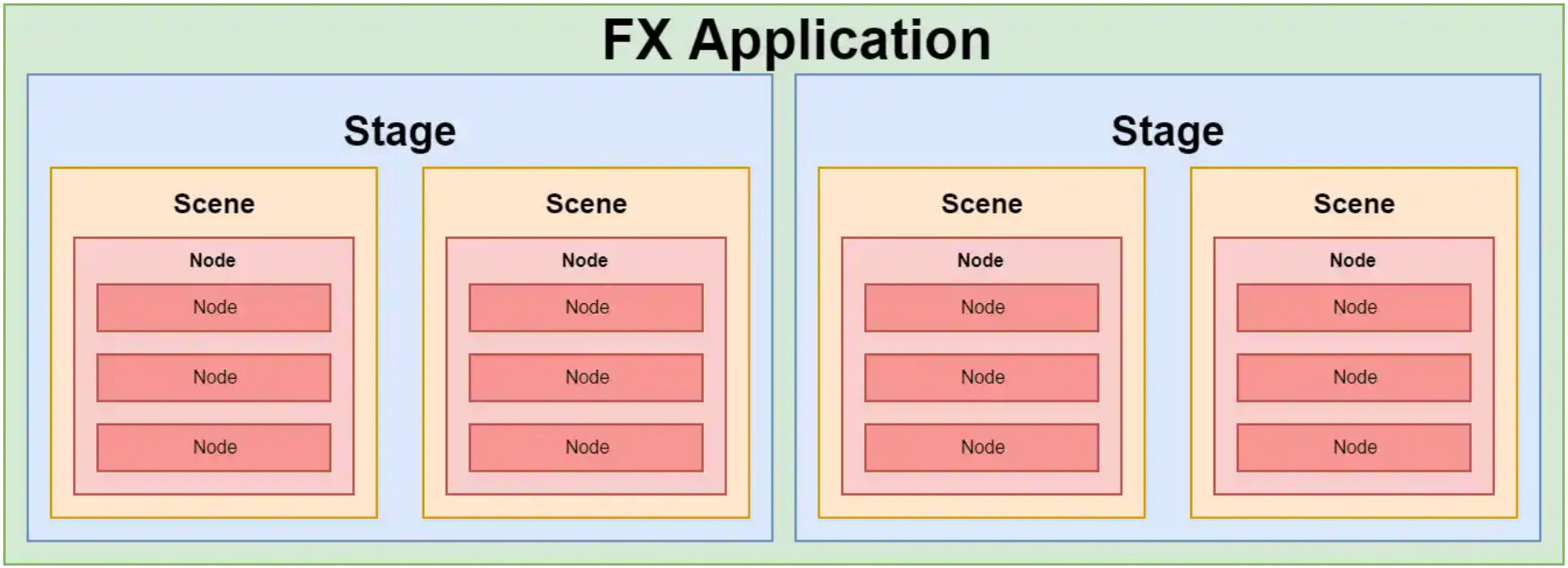
1. Avvio del runtime JavaFX
2. Istanziazione di `App` (la classe specificata che estende `Application`)
3. Invocazione metodo `start(javafx.stage.Stage)`
4. Attesa terminazione applicazione
   - mediante `Platform.exit()`
   - chiusura dell'ultima finestra (e `Platform.isImplicitExit()` è true)
5. Invocazione metodo `stop()`

Elementi fondamentali (3/4)

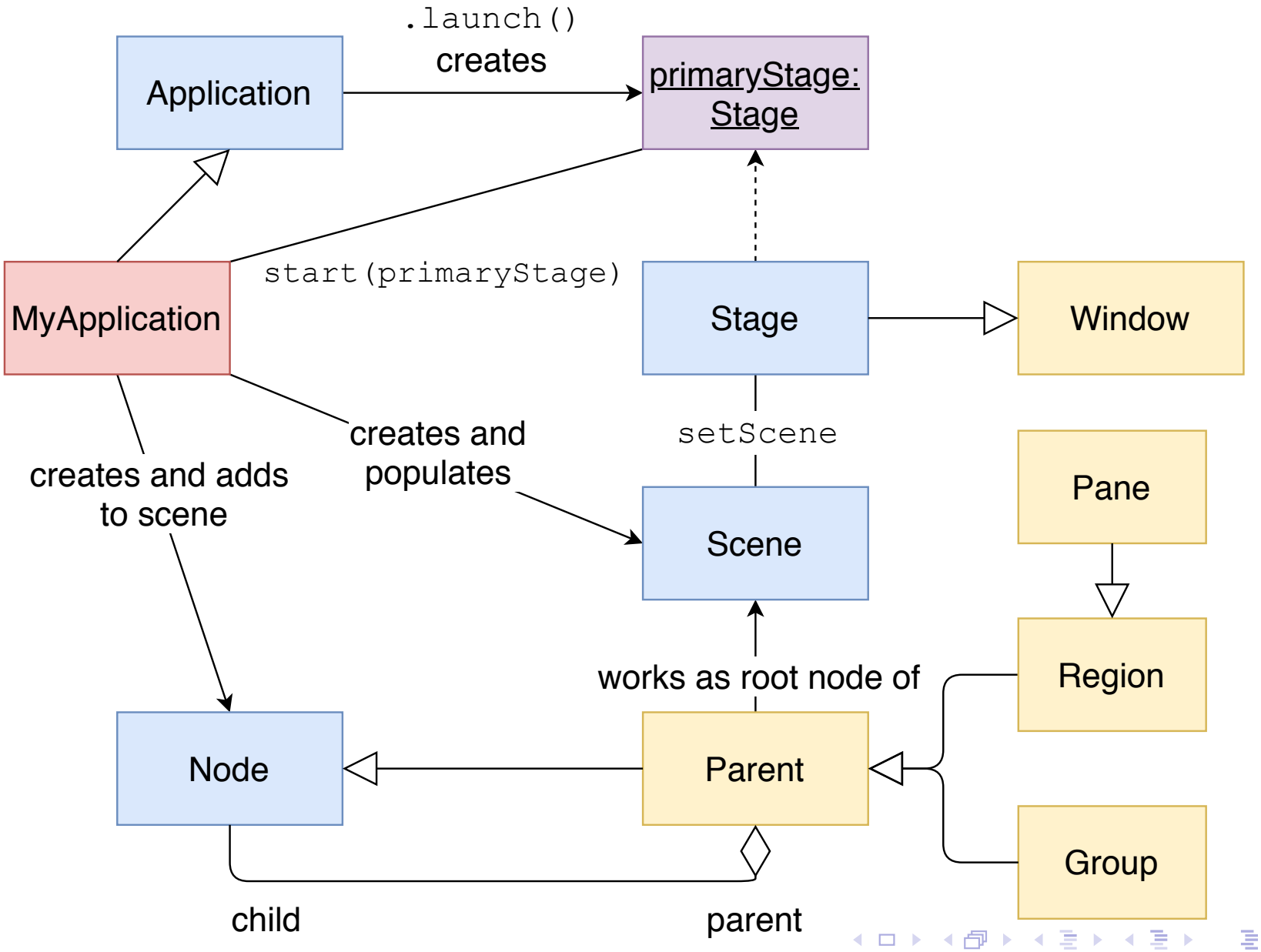
Node(s)

- Un **nodo** è un elemento/componente della scena
- Ciascun nodo ha sia la parte di view (aspetto) sia la parte di controller (comportamento)
- Hanno **proprietà** (con supporto al *binding*) e possono generare **eventi**
- Possono essere organizzati gerarchicamente
  - La sottoclasse `Parent` rappresenta nodi che possono avere figli (recuperabili via `getChildren()`)
- Un nodo ha un ID univoco, coordinate locali, può subire trasformazioni (ad es. rotazione), ha un bounding rectangle associato, e può essere stilizzato via CSS
- `https://openjfx.io/javadoc/15/javafx.graphics/javafx/scene/Node.html`

Elementi fondamentali (4/4)



Struttura di un'applicazione JavaFX-based



Linee guida[1]

1. La classe principale di un'applicazione JavaFX deve estendere la classe `javafx.application.Application`

2. Il metodo `main()` deve chiamare il metodo `launch()`
   - Si tratta di un metodo statico della classe `Application`

3. Il metodo `void start(Stage primaryStage)` è, di fatto, l'entry point dell'applicazione JavaFX (lo stage primario è creato dalla piattaforma)

4. La scena definita per lo stage (vedi metodo `setScene()`) costituisce il container principale per tutti i componenti della GUI

[1] Si faccia riferimento al repository di esempio:
`https://github.com/APICe-at-DISI/sample-javafx-project`

Nodi e Proprietà

- Ogni scena può essere popolata con una gerarchia di nodi
- Ciascun nodo (componente) espone diverse proprietà
  - relative all'aspetto (es. `size`, `posizion`, `color`, ...)
  - relative al contenuto (es. `text`, `value`, ...)
  - relative al comportamento (es. *event handler*, `controller`, ...)
- Ciascun nodo genera eventi in relazione ad azioni dell'utente

GUI con bottone e label

```
1  public class Example1 extends Application {
2      @Override
3      public void start(Stage stage) throws Exception {
4          Label lbl = new Label();
5          lbl.setText("Label text here...");
6
7          Button btn = new Button();
8          btn.setText("Click me");
9
10         HBox root = new HBox();
11         root.getChildren().add(btn);
12         root.getChildren().add(lbl);
13
14         stage.setTitle("JavaFX - Example 1");
15         stage.setScene(new Scene(root, 300,250));
16         stage.show();
17     }
18 }
```

Binding e proprietà

- Per **binding** si intende il meccanismo che consente di collegare due proprietà fra loro, in modo unidirezionale o bidirezionale
- Una `Property<T>` è un `ObservableValue<T>` che può essere collegato/scollegato ad altri osservabili o proprietà attraverso
  - `bind(ObservableValue<? extends T> observable)`
  - `bindBidirectional(Property<T> other)`
  - `unbind()`
  - `unbindBidirectional(Property<T> other)`

```
1 final TextField input = new TextField();
2 final Label mirror = new Label();
3 // connette la label con il valore del textfield
4 mirror.textProperty()
5     .bindBidirectional(input.textProperty());
```

Layouts (1/3)

Group

- Non impone nessun posizionamento per i componenti figli
- Da utilizzare per posizionare i componenti figli in posizioni fisse

Region

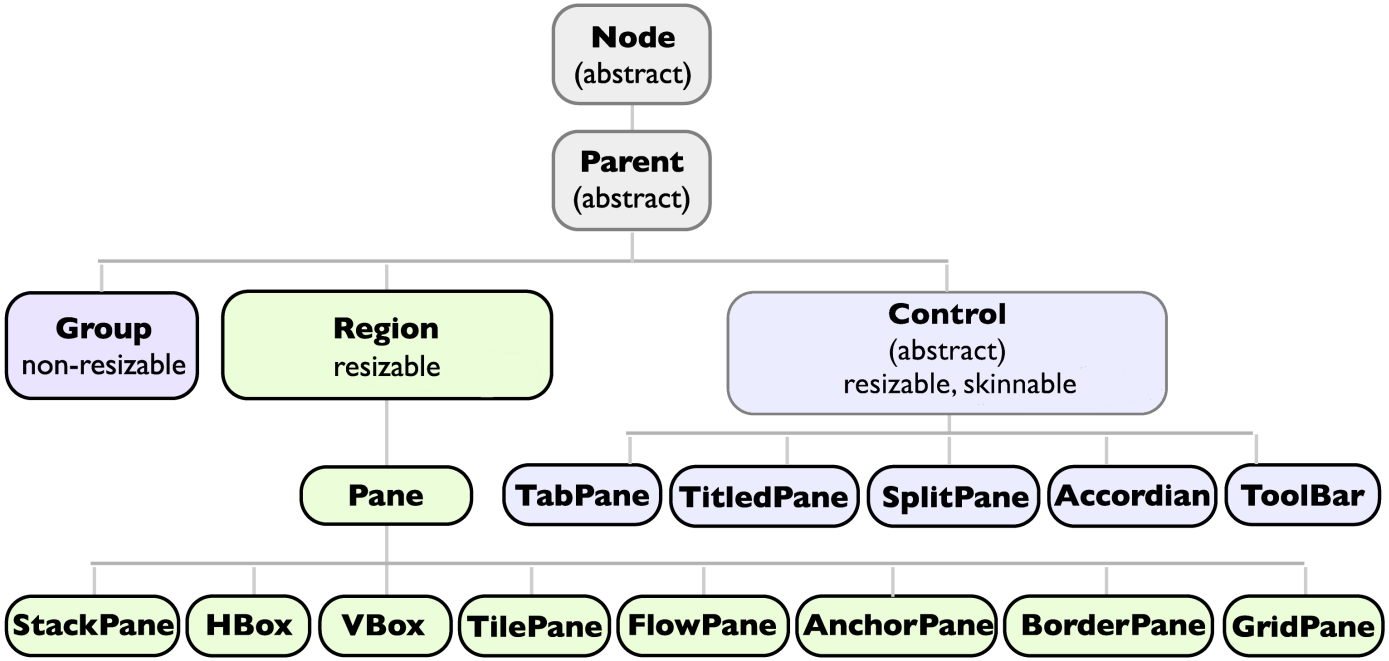
- Tutte le sue specializzazioni forniscono diversi layout general purpose
- Sono simili a quelli offerti da Swing

Control

- Costituisce l'insieme dei layout personalizzabili
- Ciascun layout di questo tipo fornisce specifiche API per l'aggiunta dei componenti figli

`https://openjfx.io/javadoc/13/javafx.graphics/javafx/scene/layout/package-summary.html`

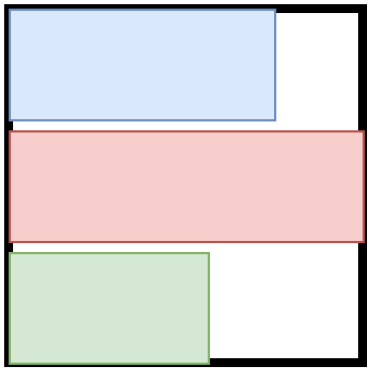
Layouts (2/3)



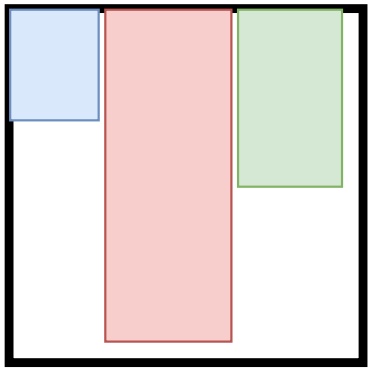
Aggiungere componenti ad un layout

- Il metodo `ObservableList<Node> getChildren()` restituisce la lista di nodi figli di un qualunque nodo/layout
- Alla lista possono essere aggiunti (`boolean add(Node e)`) e gestiti i componenti figli

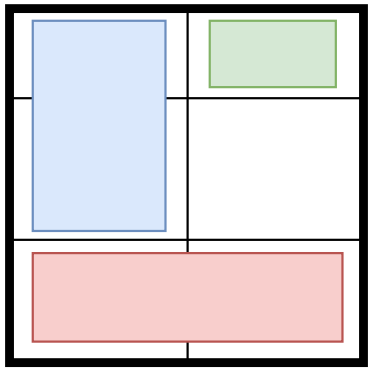
Layouts (3/3)



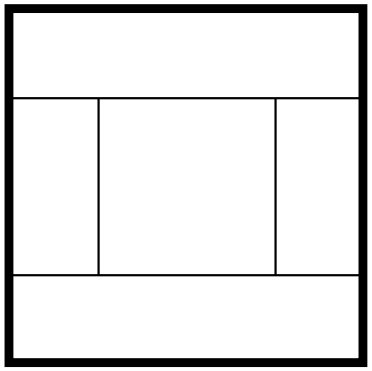
VBox



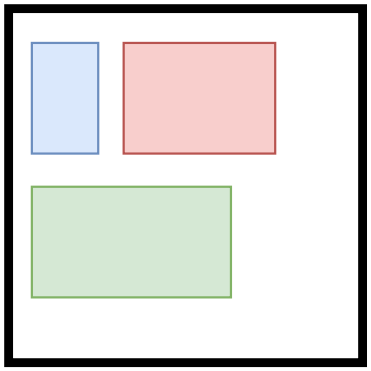
HBox



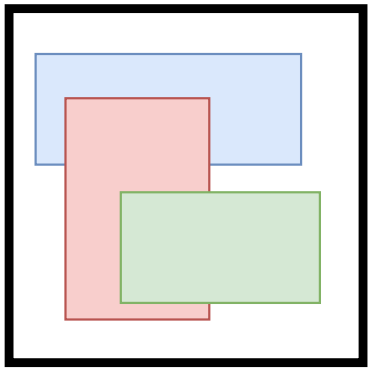
GridPane



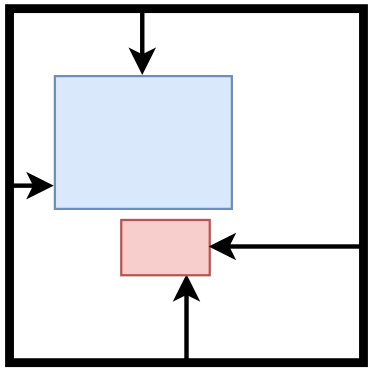
BorderPane



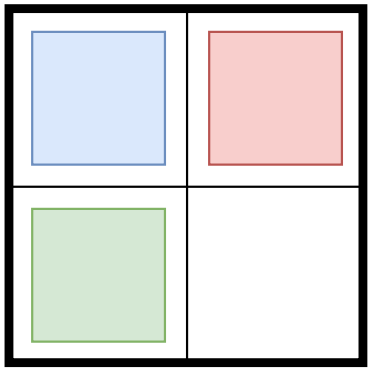
FlowPane



StackPane



AnchorPane



TilePane

Eventi

- Possono essere generati in relazione nodi e alle scene
  - Fanno riferimento alla classe `javafx.event.Event`
- Come in swing, si generano in funzione di azioni dell'utente sulla GUI
- Possono essere gestiti attraverso *event handlers* (devono implementare l'interfaccia `EventHandler`)
- Ogni nodo può registrare uno o più event handlers
  - In generale, attraverso i metodi `setOn...()`
  - Ogni event handler deve implementare il metodo `void handle(ActionEvent e)`

Es. Gestione del click su un Button Node

```
1 btn.setOnMouseClicked(event -> {
2     lbl.setText("Hello, JavaFX World!");
3 });
```

Esempio con più Stage (1/2)

```
1  public class App extends Application {
2
3    @Override
4    public final void start(final Stage mainStage) {
5      final Scene scene = new Scene(initSceneUI());
6      mainStage.setScene(scene);
7      mainStage.setTitle("JavaFX Example");
8      mainStage.show();
9    }
10
11   private Parent initSceneUI() {
12     final Label inputLbl = new Label("Input: ");
13     final TextField inputArea = new TextField();
14     final Button okBtn = new Button("Open a new Stage with the input data!");
15
16     okBtn.setOnMouseClicked(event -> {
17       new SecondStage(inputArea.getText()).show();
18     });
19
20     final BorderPane root = new BorderPane();
21     root.setRight(okBtn);
22     root.setLeft(inputLbl);
23     root.setCenter(inputArea);
24
25     BorderPane.setAlignment(inputLbl, Pos.CENTER_LEFT);
26     BorderPane.setAlignment(okBtn, Pos.CENTER_RIGHT);
27
28     return root;
29   }
30 }
```

Esempio con più Stage (2/2)

```
1  public class SecondStage extends Stage {
2      private Label lbl;
3
4      public SecondStage(final String message) {
5          super();
6          setTitle("New Window...");
7          setScene(new Scene(initSceneUI(), 400, 200));
8          lbl.setText(message);
9      }
10
11     private Parent initSceneUI() {
12         lbl = new Label();
13         FlowPane root = new FlowPane();
14         root.setAlignment(Pos.CENTER);
15         root.getChildren().add(lbl);
16         return root;
17     }
18 }
19
20 public class Main {
21     public static void main(final String[] args) {
22         Application.launch(App.class, args);
23     }
24 }
```

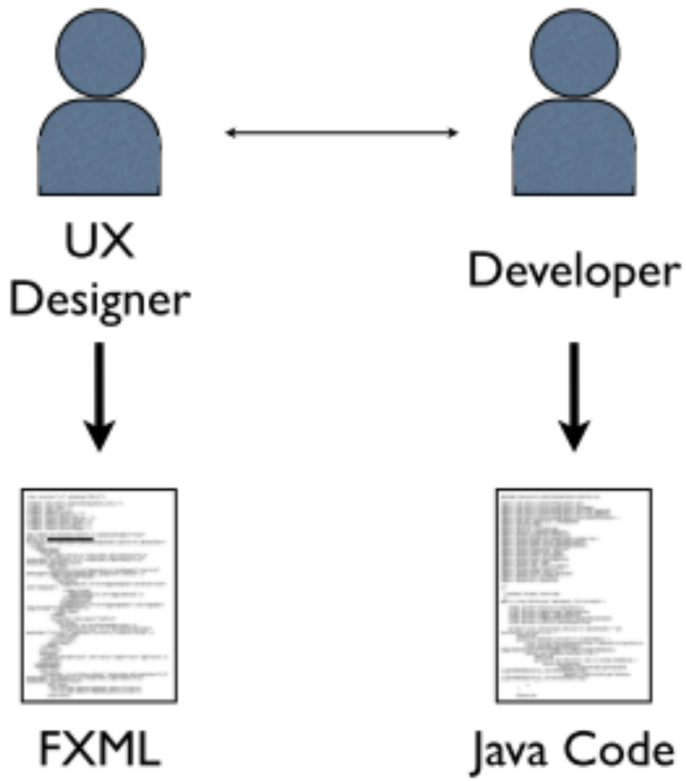
JavaFX e concorrenza

- Similarmente a Swing, JavaFX ha un singolo thread che gestisce il processing degli eventi: **JavaFX Application Thread** (JFXAT)
- Tutte le modifiche allo *scene graph* devono essere effettuate su JFXAT
- Nota: è opportuno conoscere quali metodi hook dell'`Application` sono eseguiti (ad es. `start`) oppure no (ad es. `init`) su JFXAT
- **Platform.runLater(Runnable)** accoda il runnable nella coda degli eventi del JFXAT

Outline

Separazioni di ruoli e contenuti

- In JavaFX è possibile separare il design della GUI dal codice sorgente che la riguarda
- Il design della GUI può essere descritto attraverso un linguaggio di markup denominato FXML



FXML

- Linguaggio di markup basato su XML
- Descrive la struttura della GUI
  - Tutti i componenti della GUI sono specificati mediante tag specifici
  - Le proprietà sono specificate come attributi su ciascun tag, nella forma chiave-valore
- Ogni file FXML (con estensione `.fxml`) deve essere un file XML valido
  - Deve iniziare con il tag: `<?xml version="1.0" encoding="UTF-8"?>`

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>
```

Esempio di GUI in FXML – Note

1. Attraverso il tag `<?import ... ?>` è possibile specificare i package in cui recuperare le classi dei componenti d'interesse
   - E' equivalente all'`import` di Java

2. Il container principale (unico per il singolo file) deve specificare gli attributi `xmlns` e `xmlns:fx`
   - `xmlns="http://javafx.com/javafx"`
   - `xmlns:fx="http://javafx.com/fxml"`

3. Ogni container deve specificare i nodi figli all'interno dei tag `<children>` e `</children>`

4. Ogni nodo deve definire il proprio ID mediante l'attributo `fx:id`
   - Es. `<TextField fx:id="textField1"/>`

Collegare il design della GUI al codice Java

- La GUI descritta nel file FXML deve essere collegata alla scena agganciata allo stage dell'applicazione
- Si può utilizzare il componente `javafx.fxml.FXMLLoader`
  - Il metodo statico `load(URL location)`
- Nota: occorre dichiarare il modulo `javafx.fxml` (si veda ad es. la build Gradle più avanti)

FXMLLoader (esempio)

- Si suppone che nel progetto sia presente il file `main.fxml` contenente una descrizione valida per la GUI da caricare

```
1 Parent root = FXMLLoader.load(
2   ClassLoader.getSystemResource("layouts/main.fxml"));
```

FXMLLoader (esempio completo)

```
1  public class Example3 extends Application {
2
3      @Override
4      public void start(Stage stage) throws Exception {
5          Parent root = FXMLLoader.load(ClassLoader.
       getSystemResource("layouts/main.fxml"));
6
7          Scene scene = new Scene(root, 500, 250);
8
9          stage.setTitle("JavaFX - Example 3");
10         stage.setScene(scene);
11         stage.show();
12     }
13
14     public static void main(String[] args) {
15         launch(args);
16     }
17 }
```

Lookup dei componenti della GUI

- Il riferimento ai componenti (nodi) inseriti nella GUI definita nel file FXML può essere recuperato tramite la scena a cui la GUI è stata collegata
  - Metodo `Node lookup(String id)`

Node Lookup (esempio)

```
1 Label lbl = (Label) scene.lookup("#lbl");
2
3 Button btn = (Button) scene.lookup("#btn");
4 btn.setOnMouseClicked(handler -> {
5     lbl.setText("Hello, FXML!");
6 });
```

- **Attenzione**: il metodo `lookup` richiede come parametro l'id specificato per il componente (attributo `fx:id` nel file FXML) preceduto dal simbolo #

GUI Controller e Node Injection

- Per una corretta separazione dei contenuti (e una buona implementazione del pattern MVC in JavaFX) è opportuno specificare un oggetto *controller* per ciascuna GUI
  - Il parent component della GUI deve definire l'attributo `fx:controller` con valore riferito al nome pienamente qualificato della classe che fungerà da controller

- Mediante l'annotazione `@FXML` è possibile recuperare:
  - I riferimenti ai vari nodi
    - senza utilizzare esplicitamente il meccanismo di lookup—usando la corrispondenza tra l'ID del nodo nel file FXML e il nome della variabile d'istanza annotata nella classe controller
  - Associare gli event handler ai vari eventi dei componenti

Esempio Completo (1/3) – Application

```
1  public class CompleteExample extends Application {
2
3      @Override
4      public void start(Stage stage) throws Exception {
5          VBox root = FXMLLoader.load(ClassLoader.
       getSystemResource("layouts/main.fxml"));
6
7          Scene scene = new Scene(root, 500, 250);
8
9          stage.setTitle("JavaFX - Complete Example");
10         stage.setScene(scene);
11         stage.show();
12     }
13
14     public static void main(String[] args) {
15         launch(args);
16     }
17 }
```

Esempio Completo (2/3) – GUI (FXML file)

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <?import javafx.scene.control.*?>
4  <?import javafx.scene.layout.*?>
5
6  <VBox
7      xmlns="http://javafx.com/javafx"
8      xmlns:fx="http://javafx.com/fxml"
9      fx:controller="it.unibo.oop.lab.javafx.UIController">
10   <children>
11     <Button fx:id="btn"
12         alignment="CENTER"
13         text="Say Hello!"
14         onMouseClicked="#btnOnClickHandler" />
15
16     <Label fx:id="lbl"
17         alignment="CENTER_LEFT"
18         text="Label Text Here!" />
19   </children>
20 </VBox>
```

Esempio Completo (3/3) – GUI Controller

```
1  public class UIController {
2
3      @FXML
4      private Label lbl;
5
6      @FXML
7      private Button btn;
8
9      @FXML
10     public void btnOnClickHandler() {
11         lbl.setText("Hello, World!");
12     }
13 }
```

Outline

Integrare JavaFX e Swing

- L'integrazione può avvenire nelle due direzioni
  - Si possono includere elementi Swing in applicazioni JavaFX attraverso `SwingNode`
  - Si possono includere elementi JavaFX in applicazioni Swing attraverso `JFXPanel`
  - Nota: `SwingNode` e `JFXPanel` si trovano nel modulo `javafx.swing`
- Va prestata particolare attenzione a dove viene eseguito il codice che gestisce la GUI
  - `javafx.application.Platform.runLater()`, per eseguire codice nel thread dedicato a JavaFX
  - `javax.swing.SwingUtilities.invokeLater()`, per eseguire codice nel thread dedicato a Swing

Usare JavaFX in applicazioni Swing: esempio I

```
1 public static void main(final String[] args){
2   initMainJFrame(new JFrame("JFrame GUI"));
3 }
```

```
1  private static void initMainJFrame(final JFrame frame) {
2    final JButton button = new JButton();
3    button.setText("Launch JavaFX Scene");
4    button.addActionListener(event -> {
5      final JFXPanel jfxPanel = new JFXPanel();
6      Platform.runLater(() -> {
7        jfxPanel.setScene(new Scene(initJavaFXSceneUI(), 300, 300));
8        SwingUtilities.invokeLater(() -> {
9          final JFrame frameWithJavaFX = new JFrame("JFrame with JavaFX embedded!");
10         frameWithJavaFX.add(jfxPanel);
11         frameWithJavaFX.pack();
12         frameWithJavaFX.setVisible(true);
13   }); }); });
14
15   final JPanel panel = new JPanel();
16   panel.setLayout(new FlowLayout());
17   panel.add(button);
18
19   frame.setContentPane(panel);
20   frame.setSize(300, 300);
21   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22   frame.setVisible(true);
23 }
```

Usare JavaFX in applicazioni Swing: esempio II

```
1  private static Parent initJavaFXSceneUI() {
2    final Label lbl = new Label();
3    lbl.setText("Hello, JavaFX World!");
4
5    final Button btn = new Button();
6    btn.setText("Say Hello");
7    btn.setOnMouseClicked(event -> {
8      lbl.setText("Hello from Button!");
9    });
10
11   final VBox root = new VBox();
12   root.getChildren().add(lbl);
13   root.getChildren().add(btn);
14
15   return root;
16 }
```

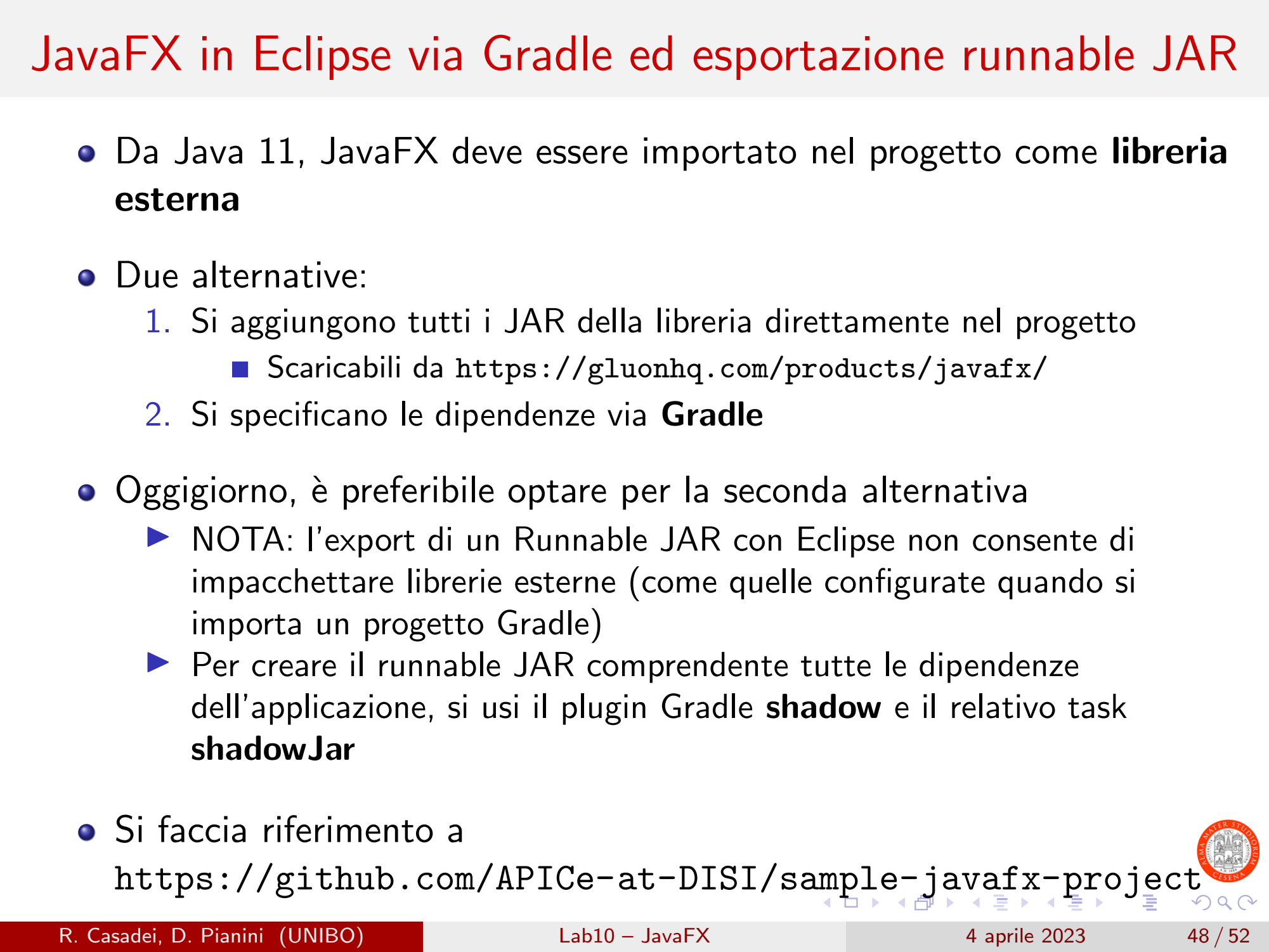
Usare Swing in applicazioni JavaFX

```
1  public final class JavaFXAppWithSwing extends Application {
2      @Override
3      public void start(final Stage primaryStage) throws Exception {
4          final SwingNode msg = new SwingNode();
5          SwingUtilities.invokeLater(() ->
6              msg.setContent(new JLabel("Hello by Swing JLabel")));
7          HBox pane = new HBox();
8          pane.getChildren().add(msg);
9          primaryStage.setScene(new Scene(pane));
10         primaryStage.show();
11     }
12
13     // ...
14 }
```

Outline

JavaFX in Eclipse via Gradle ed esportazione runnable JAR

- Da Java 11, JavaFX deve essere importato nel progetto come **libreria esterna**
- Due alternative:
  1. Si aggiungono tutti i JAR della libreria direttamente nel progetto
     - Scaricabili da `https://gluonhq.com/products/javafx/`
  2. Si specificano le dipendenze via **Gradle**
- Oggigiorno, è preferibile optare per la seconda alternativa
  - NOTA: l'export di un Runnable JAR con Eclipse non consente di impacchettare librerie esterne (come quelle configurate quando si importa un progetto Gradle)
  - Per creare il runnable JAR comprendente tutte le dipendenze dell'applicazione, si usi il plugin Gradle **shadow** e il relativo task **shadowJar**
- Si faccia riferimento a `https://github.com/APICe-at-DISI/sample-javafx-project`



build.gradle.kts (completo – sintassi Kotlin)

```
plugins {
    java // add support for Java
    // Apply the application plugin to add support for building a CLI application
    // You can run your app via task "run": ./gradlew run
    application
    // Adds task 'shadowJar' to export a runnable jar.
    // The runnable jar will be found in build/libs/projectname-all.jar
    id("com.github.johnrengelman.shadow") version "5.2.0"
}

repositories { mavenCentral() }

val javaFXModules = listOf( "base", "controls", "fxml", "swing", "graphics")
val supportedPlatforms = listOf("linux", "mac", "win") // All required for OOP
val javaFxVersion = 15

dependencies {
    for (platform in supportedPlatforms) {
        for (module in javaFXModules) {
            implementation("org.openjfx:javafx-$module:$javaFxVersion:$platform")
        }
    }
    // JUnit API and testing engine
    testImplementation("org.junit.jupiter:junit-jupiter-api:5.5.2")
    testRuntimeOnly("org.junit.jupiter:junit-jupiter-engine:5.5.2")
}

tasks.withType<Test> { useJUnitPlatform() } // Enables JUnit 5 Jupiter module

application { mainClassName = "it.unibo.samplejavafx.App" }
```

Outline

Scene Builder 2.0

- Strumento per la creazione di GUI JavaFX-based in modalità drag-n-drop (GUI Builder)
- Consente di esportare il file FXML relativo alla GUI disegnata
- Distribuito come strumento esterno al JDK, non integrato (direttamente) in Eclipse
- `https://gluonhq.com/products/scene-builder/`

Scene Builder 2.0