Introduction to GUI Programming

- GUI (Graphical User Interface): A visual way for users to interact with a computer program, using elements like windows, buttons, menus, and text fields.
- Why GUIs?
  - User-Friendly: Easier and more intuitive than command-line interfaces.
  - Interactive: Provide immediate feedback to user actions.
  - Visual Appeal: Enhance the user experience.
- Java's GUI Frameworks:
  - AWT (Abstract Window Toolkit): Oldest, platform-dependent.
  - Swing: More robust, platform-independent (pure Java), but can look dated.
  - JavaFX: Modern, rich-client platform for building cross-device applications. Successor to Swing, offering better graphics, media, and CSS styling.

What is JavaFX?

- A comprehensive set of graphics and media packages for creating rich client applications.
- Key Features:
  - Rich UI Controls: Buttons, text fields, tables, trees, etc.
  - Powerful Graphics: 2D and 3D shapes, effects, transformations.
  - Media Capabilities: Audio and video playback.
  - CSS Styling: Customize application look and feel using CSS.
  - FXML: An XML-based language for defining UI structure separately from code.
  - High Performance: Leverages hardware acceleration.
  - Cross-Platform: Runs on Windows, macOS, Linux, and even some embedded systems.

The JavaFX Application Class (Entry Point)

- Every JavaFX application must extend **javafx.application.Application**.
- The Application class defines the lifecycle of a JavaFX application.
- Key methods:
  - public void **start**(Stage primaryStage): The main entry point for all JavaFX applications. It's automatically called after launch().
  - public static void **launch**(String... args): A static method that launches the application. It calls start().
  - The **primaryStage** parameter in start() is the main window of your application.

Example :

```
// File: HelloWorldApp.java
import javafx.application.Application; // Core JavaFX Application class
import javafx.stage.Stage;          // Represents the main window

public class HelloWorldApp extends Application {

    @Override // The primary entry point for all JavaFX applications.
    public void start(Stage primaryStage) {
        // 1. Set the title of the primary stage (the main window)
        primaryStage.setTitle("My First JavaFX App");

        // 2. Display the stage
        primaryStage.show();
    }

    public static void main(String[] args) {
        // 3. Launch the JavaFX application.
        // This method internally calls the start() method.
        launch(args);
    }
}
```

Example: A Minimal JavaFX Application

- This code shows the absolute basic structure: extending Application and implementing start().
- It creates a simple window (Stage) but doesn't put anything in it yet.
- To Run: Compile HelloWorldApp.java. When running from command line, you'll need the --module-path and --add-modules flags. Most IDEs handle this configuration.

Stages and Scenes (Core Containers)

- **Stage**: Represents a top-level window (like a JFrame in Swing).
  - The **primaryStage** is the main window provided by the **start**() method. You can create additional Stage objects for pop-up windows.
  - Methods: setTitle(), setWidth(), setHeight(), setScene(), show(), close().
- **Scene**: Holds all the content of a Stage (like a JPanel in Swing).
  - A Stage can only contain one Scene at a time.
  - You can create multiple scenes and switch between them on a stage.
  - Scene constructor usually takes a Node (the root of your UI hierarchy) and optionally width/height.
- Hierarchy: Application -> Stage (Window) -> Scene (Content) -> Node (UI Element)

```
// File: StageSceneExample.java
import javafx.application.Application;
import javafx.scene.Scene;        // Represents the content area
import javafx.scene.control.Button; // A simple UI control
import javafx.scene.layout.StackPane; // A basic layout container
import javafx.stage.Stage;
public class StageSceneExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        // 1. Create a UI control (e.g., a Button)
        Button btn = new Button("Click Me!");
        // 2. Create a layout pane and add the button to it.
        // StackPane is simple: it stacks nodes on top of each other, by default centering them.
        StackPane root = new StackPane();
        root.getChildren().add(btn); // Add the button to the pane
```

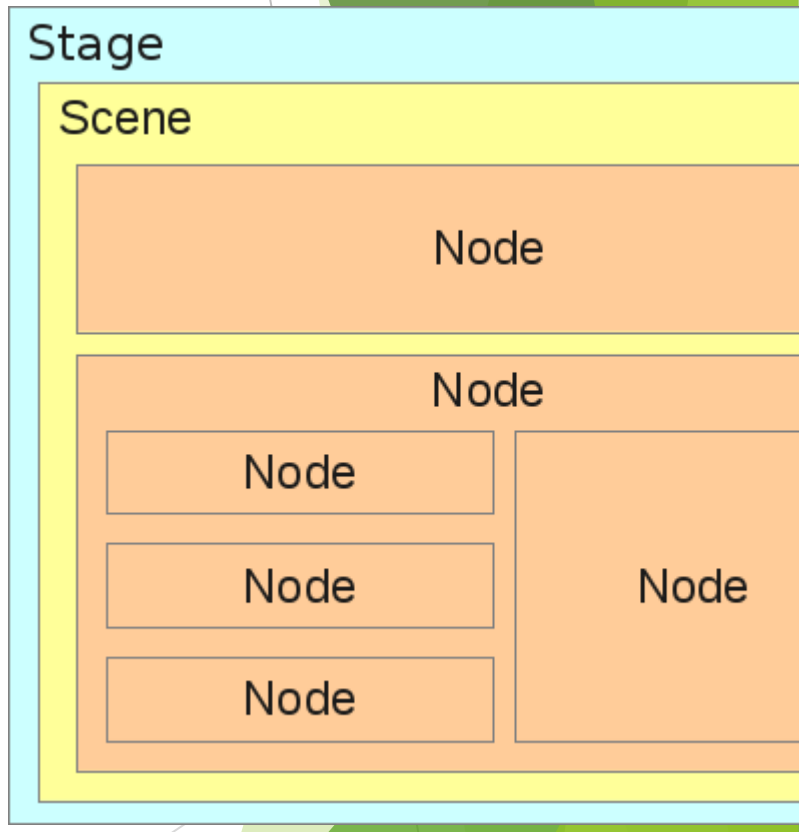
```
// 3. Create a Scene, passing the root pane and desired dimensions.
        Scene scene = new Scene(root, 300, 200); // root node, width, height
        // 4. Set the title of the stage
        primaryStage.setTitle("Stage and Scene Example");
        // 5. Set the scene on the primary stage
        primaryStage.setScene(scene);
        // 6. Display the stage
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

This example creates a simple Scene and places it on the Stage.

We're using a StackPane here, which is a basic layout pane (more on these later).

Nodes and the Scene Graph

- **Node**: The base class for all visual elements in a JavaFX Scene. Everything you see in a JavaFX application is a Node.
  - Examples: Button, Label, Rectangle, ImageView, Pane (layout containers).
- **Scene Graph**: A tree-like hierarchical data structure that represents all the nodes in a Scene.
  - The Scene has a single root node.
  - Layout panes are typically parent nodes that contain other nodes (their children).
  - UI controls and shapes are often leaf nodes.
  - Changes to nodes in the scene graph automatically update the visual display.



Layout Panes: Introduction

- Purpose: To arrange and position nodes within a Scene.
- JavaFX provides a rich set of layout panes to handle different arrangement strategies automatically. This simplifies UI design significantly.
- Common Layout Panes:
  - StackPane: Stacks nodes on top of each other, centered by default.
  - FlowPane: Arranges nodes in a flow, wrapping to the next line/column.
  - GridPane: Arranges nodes in a flexible grid of rows and columns.
  - BorderPane: Arranges nodes in five regions: top, bottom, left, right, and center.
  - HBox: Arranges nodes horizontally in a single row.
  - VBox: Arranges nodes vertically in a single column.

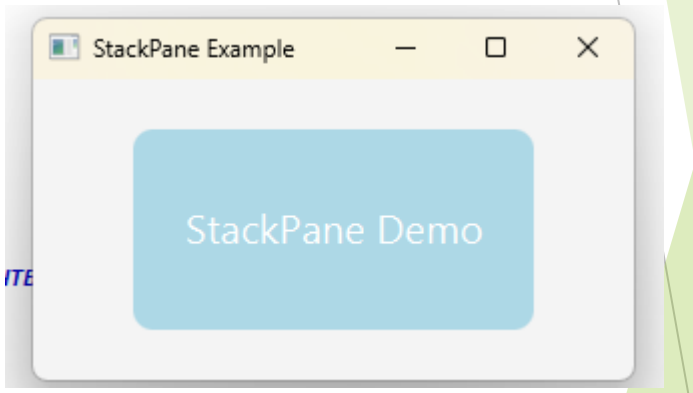
StackPane Example

- Behavior: Stacks all its children on top of each other. By default, it centers them.
- Useful for overlays or when you want a single primary component centered.
- A light blue rounded rectangle with "StackPane Demo" text centered on it.

```
// File: StackPaneExample.java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color; // For setting background color
import javafx.scene.shape.Rectangle; // A basic shape
import javafx.stage.Stage;

public class StackPaneExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        // Create a large rectangle as a background
        Rectangle backgroundRect = new Rectangle(200, 100, Color.LIGHTBLUE);
        backgroundRect.setArcWidth(20);  // Rounded corners
        backgroundRect.setArcHeight(20);
```

```
// Create a label to be stacked on top
        Label messageLabel = new Label("StackPane Demo");
        messageLabel.setStyle("-fx-font-size: 20px; -fx-text-fill: white;"); // Simple inline CSS
        // Create a StackPane and add the children (order matters for stacking)
        StackPane root = new StackPane();
        root.getChildren().addAll(backgroundRect, messageLabel); // Rectangle first, then label on top
        Scene scene = new Scene(root, 300, 150);
        primaryStage.setTitle("StackPane Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```



FlowPane Example

- Behavior: Arranges nodes in a flow, like text in a paragraph. It wraps to the next row (or column) when it runs out of space.

- Can arrange horizontally or vertically.

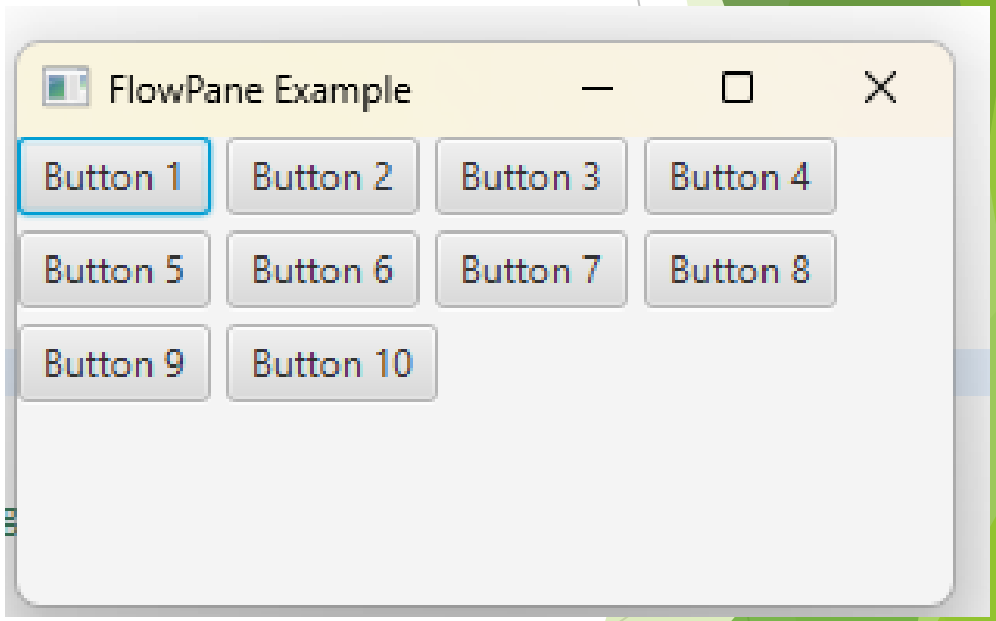
```
// File: FlowPaneExample.java
import javafx.application.Application;
import javafx.geometry.Orientation; // For FlowPane orientation
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.FlowPane;
import javafx.stage.Stage;

public class FlowPaneExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        FlowPane root = new FlowPane();
        root.setHgap(5); // Horizontal gap between nodes
        root.setVgap(5); // Vertical gap between rows/columns
        root.setOrientation(Orientation.HORIZONTAL); // Default is HORIZONTAL
```

```
        // Add multiple buttons to demonstrate flow
        for (int i = 1; i <= 10; i++) {
            root.getChildren().add(new Button("Button " + i));
        }

        Scene scene = new Scene(root, 300, 150); // Small window to force wrapping
        primaryStage.setTitle("FlowPane Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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



Buttons arranged left-to-right, top-to-bottom, wrapping to the next line if the window is too narrow.

GridPane Example

- Behavior: Lays out nodes in a flexible grid of rows and columns.
- You specify the row and column index for each child node.
- Useful for forms, calculators, or any layout requiring precise alignment.

```
// File: GridPaneExample.java
import javafx.application.Application;
import javafx.geometry.Insets; // For padding
import javafx.geometry.Pos;    // For alignment
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class GridPaneExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        GridPane root = new GridPane();
        root.setAlignment(Pos.CENTER); // Align grid in the center of the pane
        root.setPadding(new Insets(10, 10, 10, 10)); // Padding around the grid
        root.setHgap(5); // Horizontal gap between columns
        root.setVgap(5); // Vertical gap between rows
```

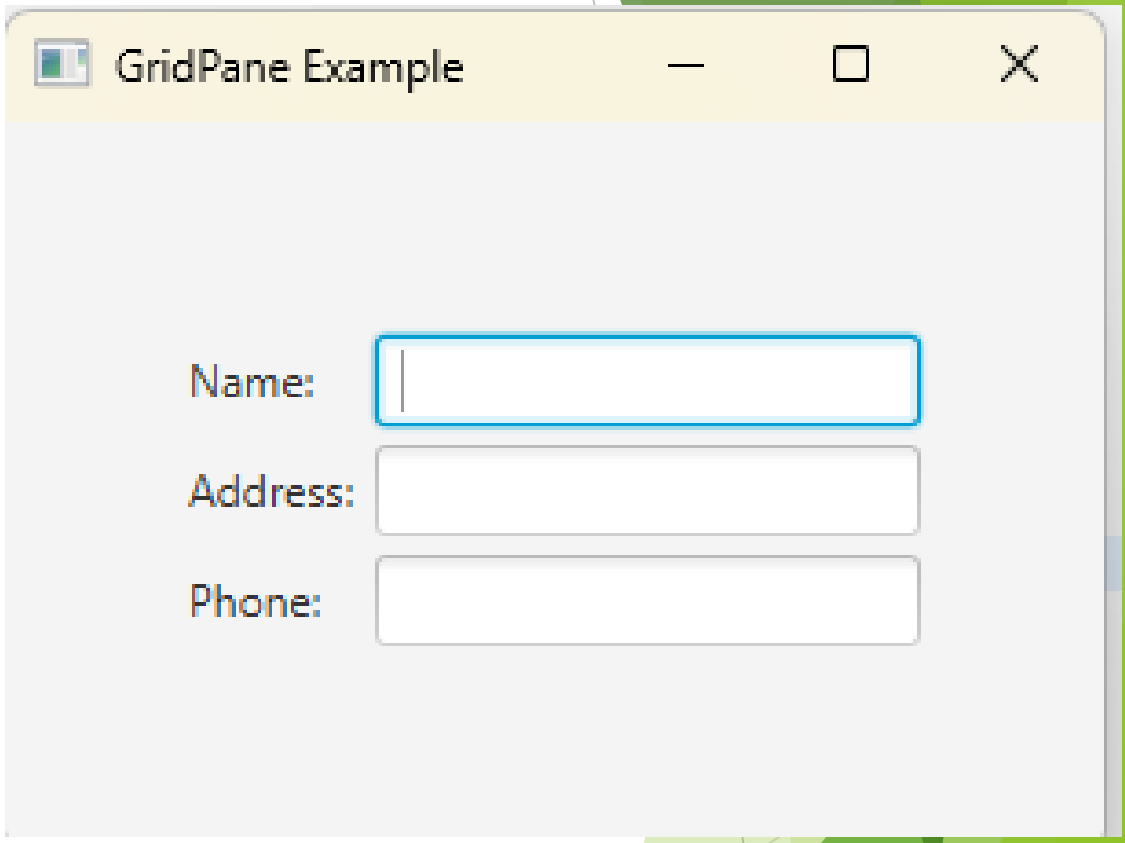
```
// Add labels and text fields to specific grid cells
        root.add(new Label("Name:"), 0, 0); // Column 0, Row 0
        root.add(new TextField(), 1, 0);    // Column 1, Row 0

        root.add(new Label("Address:"), 0, 1); // Column 0, Row 1
        root.add(new TextField(), 1, 1);      // Column 1, Row 1

        root.add(new Label("Phone:"), 0, 2); // Column 0, Row 2
        root.add(new TextField(), 1, 2);      // Column 1, Row 2

        Scene scene = new Scene(root, 300, 200);
        primaryStage.setTitle("GridPane Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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



A form-like layout with labels in one column and text fields in another, aligned by grid.

BorderPane Example

- Behavior: Divides its layout area into five regions: Top, Bottom, Left, Right, and Center.
- Only one node can be placed in each region.
- The center region expands to fill available space.

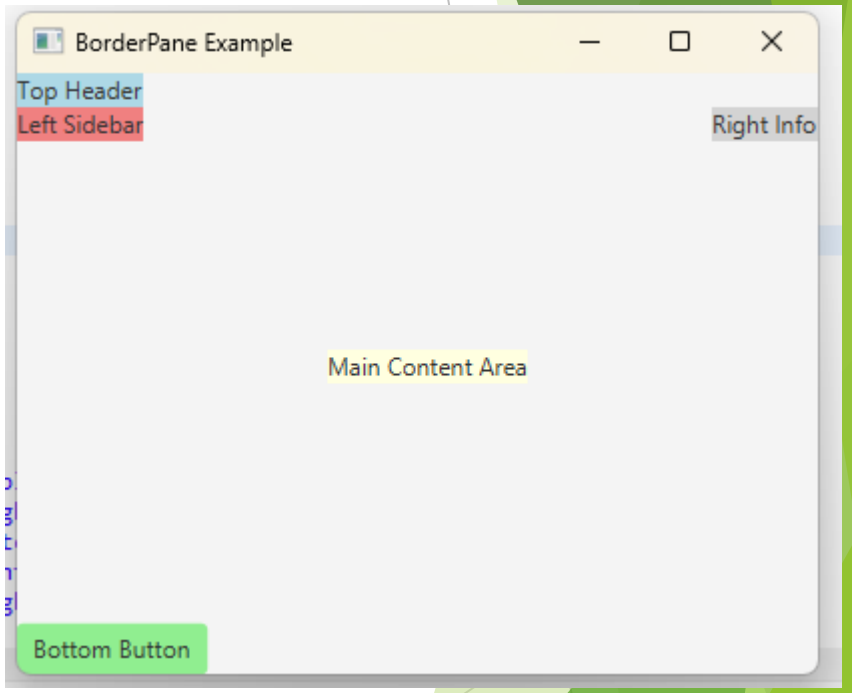
```
// File: BorderPaneExample.java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class BorderPaneExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        BorderPane root = new BorderPane();
        // Add nodes to specific regions
        root.setTop(new Label("Top Header"));
        root.setBottom(new Button("Bottom Button"));
        root.setLeft(new Label("Left Sidebar"));
        root.setRight(new Label("Right Info"));
        root.setCenter(new Label("Main Content Area"));
```

```
// (Optional) Set styles for visibility
    root.getTop().setStyle("-fx-background-color: lightblue;");
    root.getBottom().setStyle("-fx-background-color: lightgreen;");
    root.getLeft().setStyle("-fx-background-color: lightcoral;");
    root.getRight().setStyle("-fx-background-color: lightgray;");
    root.getCenter().setStyle("-fx-background-color: lightyellow;");

    Scene scene = new Scene(root, 400, 300);
    primaryStage.setTitle("BorderPane Example");
    primaryStage.setScene(scene);
    primaryStage.show();
  }

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



HBox and VBox Examples

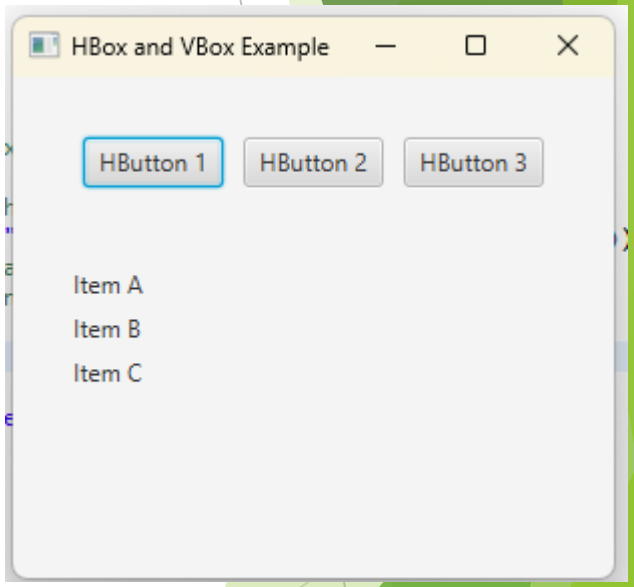
- HBox (Horizontal Box): Arranges nodes in a single horizontal row.
- VBox (Vertical Box): Arranges nodes in a single vertical column.
- Both offer properties for spacing between children and alignment.

```
// File: HBoxVBoxExample.java
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class HBoxVBoxExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        // HBox example
        HBox hbox = new HBox(10); // Spacing of 10 pixels between children
        hbox.setPadding(new Insets(10)); // Padding around the HBox
        hbox.setAlignment(Pos.CENTER); // Align children in the center of the HBox
        hbox.getChildren().addAll(
                new Button("HButton 1"),
                new Button("HButton 2"),
                new Button("HButton 3")
        );
```

```
    // VBox example
    VBox vbox = new VBox(5); // Spacing of 5 pixels between children
    vbox.setPadding(new Insets(10));
    vbox.setAlignment(Pos.TOP_LEFT); // Align children to top-left of the VBox
    vbox.getChildren().addAll(new Label("Item A"), new Label("Item B"), new Label("Item C"));
    // Nested layouts: put HBox and VBox in a parent VBox
    VBox root = new VBox(20); // Spacing between HBox and VBox
    root.setPadding(new Insets(20));
    root.getChildren().addAll(hbox, vbox);

    Scene scene = new Scene(root, 300, 250);
    primaryStage.setTitle("HBox and VBox Example");
    primaryStage.setScene(scene);
    primaryStage.show();
}
public static void main(String[] args) {
    launch(args);
}
}
```



Common UI Controls: Introduction

- JavaFX provides a rich set of pre-built UI components that enable user interaction.
- These are concrete Node types.
- Each control has specific properties and events.
- Examples:
  - Label: Displays static text or an image.
  - Button: Triggers an action when clicked.
  - TextField: Allows single-line text input.
  - CheckBox, RadioButton, ComboBox, ListView, Slider, etc. (covered in later chapters).

Label Control Example

- Used to display uneditable text, often used for captions or informational messages.
- Can also display graphics (images).

```
// Create a Label
Label helloLabel = new Label("Hello, JavaFX!");

// Set properties of the label
helloLabel.setFont(Font.font("Arial", FontWeight.BOLD, 24)); // Set font, weight, size
helloLabel.setTextFill(Color.DARKBLUE); // Set text color
helloLabel.setWrapText(true); // Allow text to wrap if it's too long

// Another label with more styling
Label styledLabel = new Label("This label is styled with CSS.");
styledLabel.setStyle("-fx-background-color: #EEE; -fx-padding: 5px; -fx-border-color: #AAA; -fx-border-width: 1px;");
```

Button Control Example

- A fundamental UI control that allows users to trigger actions.
- Its primary interaction is handling click events.

```
public void start(Stage primaryStage) {
        // Create a Button
        Button clickMeButton = new Button("Click Me!");

        // Add the button to a StackPane (for centering)
        StackPane root = new StackPane();
        root.getChildren().add(clickMeButton);

        Scene scene = new Scene(root, 200, 100);
        primaryStage.setTitle("Button Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
```

A simple window with a centered "Click Me!" button. This button doesn't do anything yet, which leads to event handling.

Handling Events: Introduction

- Events: User actions (e.g., clicking a button, typing in a text field, moving a mouse).
- Event Handling: The process of responding to these events.
- JavaFX Event Model:
  - Event Source: The object on which the event occurs (e.g., a Button is the source of a click event).
  - Event Type: Defines the type of event (e.g., ActionEvent for button clicks, MouseEvent for mouse actions, KeyEvent for keyboard actions).
  - Event Handler (or Listener): An object that implements a specific event-handling interface (e.g., EventHandler<ActionEvent>) and contains the code to execute when the event occurs.

Event Handling: Anonymous Inner Class

- A traditional way to create event handlers.
- You define an anonymous class that implements the EventHandler interface and override its handle() method.

```
public void start(Stage primaryStage) {
        Button clickMeButton = new Button("Click Me (Anon Class)");

        // Create an anonymous inner class for the event handler
        clickMeButton.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                System.out.println("Button clicked! (Anonymous Inner Class)");
            }
        });
```

Clicking the button will print "Button clicked! (Anonymous Inner Class)" to the console.

Event Handling: Lambda Expressions (Java 8+)

- A more concise and modern way to write event handlers, especially for functional interfaces (like EventHandler).
- Lambda expressions simplify the code, making it more readable.

```
public void start(Stage primaryStage) {
        Button clickMeButton = new Button("Click Me (Lambda)");

        // Use a lambda expression for the event handler
        clickMeButton.setOnAction(event -> { // event is the ActionEvent parameter
            System.out.println("Button clicked! (Lambda Expression)");
            // You can also change UI elements here, e.g.,
            // clickMeButton.setText("Clicked!");
        });
```

Same functionality as the anonymous inner class, but with cleaner syntax.

TextField Control Example with Event Handling

- Allows users to input a single line of text.
- Commonly used with its getText() and setText() methods.
- Can also handle ActionEvent when the user presses Enter in the field.

```
public void start(Stage primaryStage) {
    Label promptLabel = new Label("Enter your name:");
    TextField nameField = new TextField();
    Label greetingLabel = new Label("Hello, "); // Initial text

    // Event handler for the TextField: triggered when Enter is pressed
    nameField.setOnAction(event -> {
        String enteredName = nameField.getText(); // Get text from the TextField
        greetingLabel.setText("Hello, " + enteredName + "!"); // Update the Label
        nameField.clear(); // Clear the TextField after processing
    });
```

A window with a label, a text field for input, and another label that updates to greet the user after they type their name and press Enter.

Displaying Images (ImageView)

- The javafx.scene.image.Image class represents a graphical image.
- The javafx.scene.image.ImageView class is a Node that displays an Image.
- Supports common image formats (PNG, JPEG, GIF, BMP).

```
public void start(Stage primaryStage) {
        String imagePath = "Java_logo_icon.png";
        Image image = new Image(imagePath); // Load the image

        // Create an ImageView to display the image
        ImageView imageView = new ImageView(image);
        imageView.setFitWidth(100);   // Set preferred width
        imageView.setPreserveRatio(true); // Maintain aspect ratio

        StackPane root = new StackPane();
        root.getChildren().add(imageView);

        Scene scene = new Scene(root, 200, 200);
        primaryStage.setTitle("ImageView Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
```

Displaying Text (Text vs. Label)

- javafx.scene.text.Text: A Node specifically for displaying text. It offers more fine-grained control over text layout and rendering (e.g., advanced font metrics, line spacing). Not a control.
- javafx.scene.control.Label: A UI Control designed for simple text display, often alongside other controls. It offers wrapping and can contain ImageView.

When to use what:

- Use Label for simple captions, prompts, or static text within forms/controls.
- Use Text when you need to draw text as part of a custom graphic, or require precise control over text rendering (e.g., drawing text on a Canvas).

```
public void start(Stage primaryStage) {
        Pane root = new Pane(); // Use Pane for absolute positioning

        Text customText = new Text(50, 50, "Custom Text Node"); // X, Y, text
        customText.setFont(Font.font("Verdana", FontWeight.NORMAL, 30));
        customText.setFill(Color.PURPLE);
        customText.setStrikethrough(true); // Example of a Text-specific property

        Text positionedText = new Text("Hello World!");
        positionedText.setX(100); // Set X coordinate
        positionedText.setY(100); // Set Y coordinate
        positionedText.setFont(Font.font("Times New Roman", FontWeight.BOLD, 20));
        positionedText.setFill(Color.GREEN);

        root.getChildren().addAll(customText, positionedText);

        Scene scene = new Scene(root, 400, 200);
        primaryStage.setTitle("Text Node Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
```

Shapes: Introduction

- JavaFX provides a rich set of built-in javafx.scene.shape classes for drawing 2D vector graphics.
- These are Node types, so they can be added to any layout pane and participate in the scene graph.
- Common properties: fill (color inside), stroke (border color), strokeWidth (border thickness).
- Examples:
  - Line, Circle, Rectangle, Ellipse, Arc, Polygon, Polyline.
  - These are powerful for custom drawings and visualizations.

Circle Shape Example

- Creates a circular shape.
- Properties: centerX, centerY, radius, fill, stroke, strokeWidth.

```
public void start(Stage primaryStage) {
    Pane root = new Pane(); // Use Pane for manual positioning

    // Create a blue circle
    Circle blueCircle = new Circle(100, 75, 50); // centerX, centerY, radius
    blueCircle.setFill(Color.BLUE);
    blueCircle.setStroke(Color.DARKBLUE);
    blueCircle.setStrokeWidth(3);

    // Create a red circle with an opaque fill
    Circle redCircle = new Circle(200, 75, 40);
    redCircle.setFill(Color.RED.deriveColor(0, 1, 1, 0.5)); // 50% opacity
    redCircle.setStroke(Color.BLACK);
    redCircle.setStrokeWidth(2);

    root.getChildren().addAll(blueCircle, redCircle);

    Scene scene = new Scene(root, 300, 150);
    primaryStage.setTitle("Circle Shape Example");
    primaryStage.setScene(scene);
    primaryStage.show();
}
```

Two circles, one blue and opaque, one red and semi-transparent, drawn on the window.

Rectangle Shape Example

- Creates a rectangular shape.
- Properties: x, y, width, height, fill, stroke, strokeWidth, arcWidth, arcHeight (for rounded corners).

```
public void start(Stage primaryStage) {
        Pane root = new Pane();

        // Create a green square
        Rectangle greenRect = new Rectangle(50, 50, 80, 80); // x, y, width, height
        greenRect.setFill(Color.LIGHTGREEN);
        greenRect.setStroke(Color.DARKGREEN);
        greenRect.setStrokeWidth(2);

        // Create a rounded red rectangle
        Rectangle roundedRect = new Rectangle(150, 40, 120, 60);
        roundedRect.setFill(Color.LIGHTCORAL);
        roundedRect.setStroke(Color.BROWN);
        roundedRect.setStrokeWidth(3);
        roundedRect.setArcWidth(20); // Horizontal arc for rounded corners
        roundedRect.setArcHeight(20); // Vertical arc for rounded corners

        root.getChildren().addAll(greenRect, roundedRect);

        Scene scene = new Scene(root, 350, 180);
        primaryStage.setTitle("Rectangle Shape Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
```

A green square and a rounded red rectangle displayed on the window.

Applying CSS to JavaFX

- JavaFX supports styling applications using CSS (Cascading Style Sheets).
- This separates presentation from logic, making UIs easier to design and maintain.
- Ways to apply CSS:
  - Inline Styles: Directly on nodes using node.setStyle("-fx-property: value;"); (for quick tests).
  - Internal Stylesheets: Add CSS directly to a Scene via scene.getStylesheets().add("data:text/css,..."); (less common for large CSS).
  - External Stylesheets: Load CSS from an external .css file via scene.getStylesheets().add("path/to/style.css"); (recommended for larger projects).

Example: Simple CSS Styling (External Stylesheet)

```
public void start(Stage primaryStage) {
        Button myButton = new Button("Styled Button");
        myButton.setId("my-special-button"); // Assign an ID for CSS targeting
        StackPane root = new StackPane();
        root.getChildren().add(myButton);
        Scene scene = new Scene(root, 200, 100);
        // Link the external CSS file
        // IMPORTANT: "style.css" should be in the same directory as your .java file
        scene.getStylesheets().add(getClass().getResource("style.css").toExternalForm());
        primaryStage.setTitle("CSS Styling Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
```

```
/* style.css */
.button {
    -fx-font-size: 16px;
    -fx-background-color: #4CAF50; /* Green */
    -fx-text-fill: white;
    -fx-padding: 10px 20px;
    -fx-border-radius: 5px;
}
/* Style the button with ID "my-special-button" */
#my-special-button {
    -fx-background-color: #FF5722; /* Orange */
    -fx-font-weight: bold;
}
/* Style the button when hovered */
.button:hover {
    -fx-background-color: #607D8B; /* Blue Grey */
}
```

A button styled with CSS. The general .button style applies, and then the #my-special-button style overrides the background color, and the :hover pseudo-class changes the background on mouse-over.

Running JavaFX Applications (Revisited)

- Modular Applications (Java 9+):
  - If using a modular JDK, you often need a module-info.java file.
  - Example module-info.java for simple app:

```
module my.first.javafx.app {
    requires javafx.controls; // For Scene, Button, etc.
    requires javafx.fxml;     // If you use FXML
    opens mypackage to javafx.graphics, javafx.fxml; // Open your package to JavaFX runtime
    exports mypackage; // If other modules need to use your classes
}
```

- Compile and run with --module-path and --add-modules arguments pointing to your JavaFX SDK lib directory.
- IDE Support: Most modern IDEs (IntelliJ IDEA, Eclipse, VS Code) have excellent JavaFX support, simplifying project setup and run configurations. Refer to your IDE's documentation for precise setup instructions.

Conclusion & Key Takeaways

- JavaFX: A powerful, modern framework for building rich, cross-platform GUIs.
- Core Structure: Applications extend Application, using Stage for windows and Scene for content.
- Scene Graph: All UI elements are Nodes organized in a tree structure.
- Layout Panes: Essential for arranging nodes (e.g., StackPane, FlowPane, GridPane, BorderPane, HBox, VBox).
- UI Controls: Pre-built interactive components (Label, Button, TextField, etc.).
- Event Handling: Responding to user actions (via anonymous inner classes or lambda expressions).
- Styling: Leverage CSS for customizable look and feel.
- This chapter lays the foundational understanding for building visually compelling and interactive Java applications.