Introduction to Programming

Week 14

Magnus Madsen

Monday

- UI Programming with JavaFX

Thursday

- Live Programming

Prologue

Quote of the Week

"If you are not willing to put a fragment of your soul into the code, don't program. Do pursue a different path instead..."

— Dmitry Galuscenko

Epigram of the Week

“Every program has (at least) two purposes: the one for which it was written, and another for which it wasn’t.”

— Alan Perlis

Introduction to GUI Programming

What is a GUI?

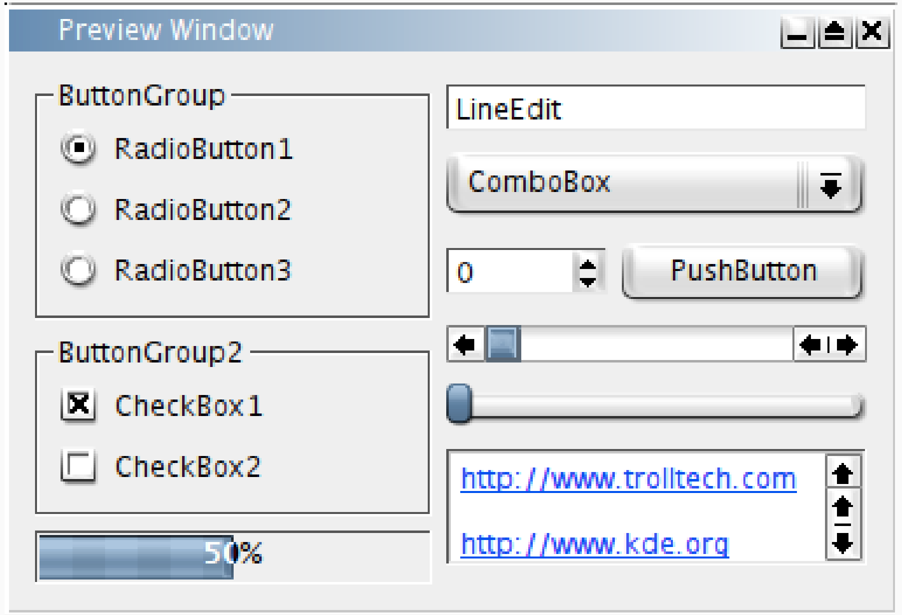
✏️ A **GUI** (graphical user interface) is a type of user interface which allows people to interact with a computer through a metaphor of direct manipulation of graphical images and widgets in addition to text.

via Wiktionary

What is a GUI?

✏️ A **GUI** (graphical user interface) is a type of user interface which allows people to interact with a computer through a metaphor of direct manipulation of graphical images and widgets in addition to text.

via Wiktionary



What is JavaFX?

JavaFX is a *modern* Java framework for building **graphical user interfaces** (GUIs).

What is JavaFX?

JavaFX is a *modern* Java framework for building **graphical user interfaces** (GUIs).

- **Platform-independent**: Write once, run on Windows, macOS, and Linux

What is JavaFX?

JavaFX is a *modern* Java framework for building **graphical user interfaces** (GUIs).

- **Platform-independent**: Write once, run on Windows, macOS, and Linux
- **Rich set of UI controls**: Buttons, text fields, tables, charts, and more

What is JavaFX?

JavaFX is a *modern* Java framework for building **graphical user interfaces** (GUIs).

- **Platform-independent**: Write once, run on Windows, macOS, and Linux
- **Rich set of UI controls**: Buttons, text fields, tables, charts, and more
- **Scene graph architecture**: Hierarchical structure for organizing UI elements

What is JavaFX?

JavaFX is a *modern* Java framework for building **graphical user interfaces** (GUIs).

- **Platform-independent**: Write once, run on Windows, macOS, and Linux
- **Rich set of UI controls**: Buttons, text fields, tables, charts, and more
- **Scene graph architecture**: Hierarchical structure for organizing UI elements
- **FXML**: Optional XML-based markup for defining UI layouts

What is JavaFX?

JavaFX is a *modern* Java framework for building **graphical user interfaces** (GUIs).

- **Platform-independent**: Write once, run on Windows, macOS, and Linux
- **Rich set of UI controls**: Buttons, text fields, tables, charts, and more
- **Scene graph architecture**: Hierarchical structure for organizing UI elements
- **FXML**: Optional XML-based markup for defining UI layouts
- **Built-in animations**: Support for transitions and effects

What is JavaFX?

JavaFX is a *modern* Java framework for building **graphical user interfaces** (GUIs).

- **Platform-independent**: Write once, run on Windows, macOS, and Linux
- **Rich set of UI controls**: Buttons, text fields, tables, charts, and more
- **Scene graph architecture**: Hierarchical structure for organizing UI elements
- **FXML**: Optional XML-based markup for defining UI layouts
- **Built-in animations**: Support for transitions and effects

i *modern* = 2008

Evolution of Java GUI frameworks:

- **AWT (1995)**: Abstract Window Toolkit - Java's first GUI toolkit
  - Limited set of components
  - Platform-dependent look and feel

History and Evolution

Evolution of Java GUI frameworks:

- **AWT (1995)**: Abstract Window Toolkit - Java's first GUI toolkit
  - Limited set of components
  - Platform-dependent look and feel
- **Swing (1998)**: Lightweight, pure Java components
  - More components and flexibility
  - Pluggable look and feel
  - Still widely used in legacy applications

Evolution of Java GUI frameworks:

- **AWT (1995)**: Abstract Window Toolkit - Java's first GUI toolkit
  - Limited set of components
  - Platform-dependent look and feel
- **Swing (1998)**: Lightweight, pure Java components
  - More components and flexibility
  - Pluggable look and feel
  - Still widely used in legacy applications
- **JavaFX (2008)**: Modern replacement for Swing
  - **Hardware-accelerated graphics**
  - **Rich multimedia support**
  - **Modern UI patterns and controls**

Event-Driven Programming

i JavaFX is based on **event-driven programming** i JavaFX is based on **event-driven programming**

Traditional (Sequential)

- Program controls the flow
- Execute statements in order
- Program asks for input when needed

> i JavaFX is based on **event-driven programming**

Traditional (Sequential)

- Program controls the flow
- Execute statements in order
- Program asks for input when needed

Event-Driven

- User controls the flow through **events**
- Program *waits* for user actions
- Program *responds* to events like:
  - Mouse clicks
  - Key presses
  - Window resizing
  - Timer expiration

How to Create a JavaFX Project in Intellij IDEA

Steps to create a new JavaFX project:

1. **File → New → Project**
2. **Select JavaFX** from the left panel
   - Choose your Project SDK (Java 11 or higher)
   - Click Next
3. **Configure Project:**
   - Enter project name
   - Choose project location
   - Select build system: **Maven** or **Gradle** (recommended)
4. **Dependencies are automatically added.**

IntelliJ creates a ready-to-run JavaFX project with all configurations!

JavaFX Fundamentals

Application Lifecycle

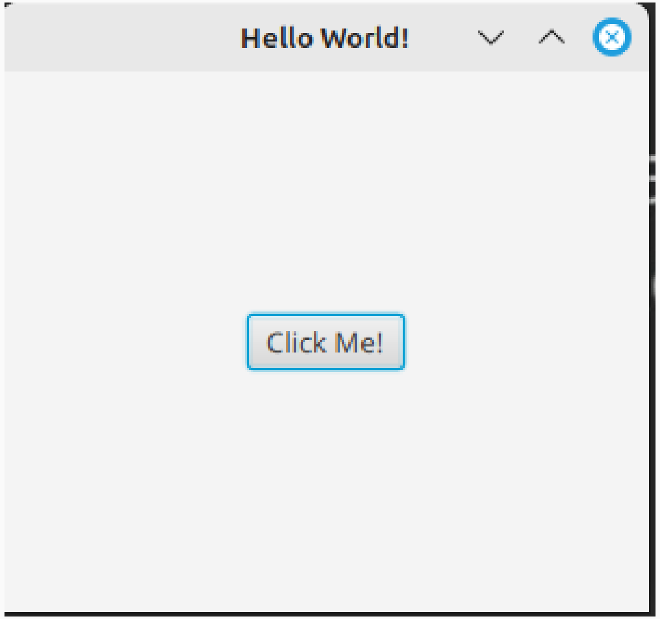
Every JavaFX application follows a specific lifecycle:

1. **init()**: Initialize resources (called once)
2. **start()**: Create and show the UI (called once)
3. **stop()**: Clean up resources (called once)

```
public class MyApp extends Application {
    public void init() {
        // Load configuration, connect to database
    }

    public void start(Stage primaryStage) {
        // Build and display UI
    }

    public void stop() {
        // Save state, close connections
    }
}
```

Application Lifecycle: Minimal Example

```java
public class HelloWorld extends Application {
    public void start(Stage primaryStage) {
        Button button = new Button("Click Me!");

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

        Scene scene = new Scene(root, 300, 250);
        primaryStage.setTitle("Hello World!");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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



Stage and Scene Concepts

- **Stage**: The window that displays your application
  - JavaFX automatically provides a primary stage when your application starts.
  - You can create additional stages to display multiple windows.
- **Scene**: The container that holds all your user interface content
  - Every scene contains a root node that holds all other UI elements.
  - Each scene has a specified width and height in pixels.
- **Nodes**: The individual UI elements that make up your interface
  - All nodes are organized in a hierarchical tree structure called the scene graph.

A stage contains a scene, which contains various nodes.

Scene Graph Architecture

JavaFX uses a **hierarchical scene graph** to organize UI elements:

- **Tree structure** where each node can have children
  - **Root node** at the top (typically a layout container)
  - **Leaf nodes** are UI controls (buttons, labels, etc.)
  - **Branch nodes** are containers that hold other nodes

```
Scene
  └── Root (VBox)
        ├── MenuBar
        ├── ToolBar
        └── BorderPane
              ├── Center: TableView
              ├── Bottom: StatusBar
              └── Right: VBox
                    ├── Label
                    └── Button
```

Q:

What happens if you run this program?

```
public class Main extends Application {
  public void start(Stage stage) {
      Label label = new Label("Hello World");
      StackPane root = new StackPane();
      root.getChildren().add(label);

      Scene scene = new Scene(root, 300, 200);
      stage.setScene(scene);
  }
}
```

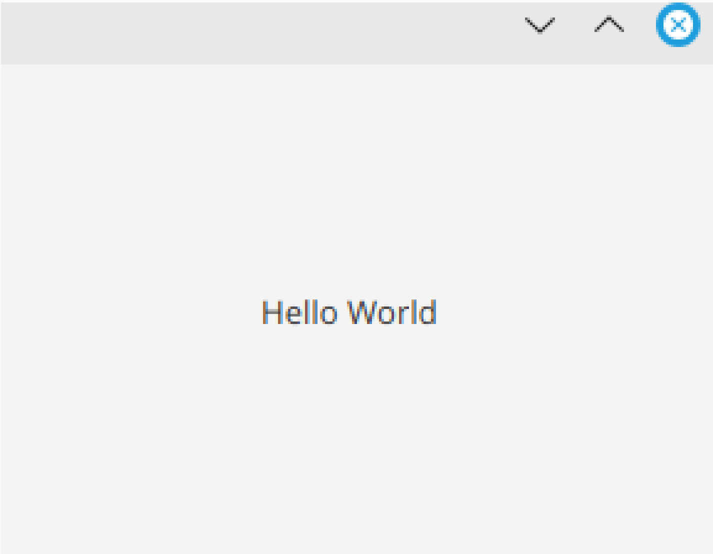
Answer

You actually get nothing! Because we did not call `show` on the stage.

Answer

You actually get nothing! Because we did not call `show` on the stage.

```
public class Main extends Application {
  public void start(Stage stage) {
      Label label = new Label("Hello World");
      StackPane root = new StackPane();
      root.getChildren().add(label);

      Scene scene = new Scene(root, 300, 200);
      stage.setScene(scene);
      stage.show();      //newly added line
  }
}
```



Basic UI Components

Summary of Basic UI Components

These are the fundamental building blocks for most user interfaces

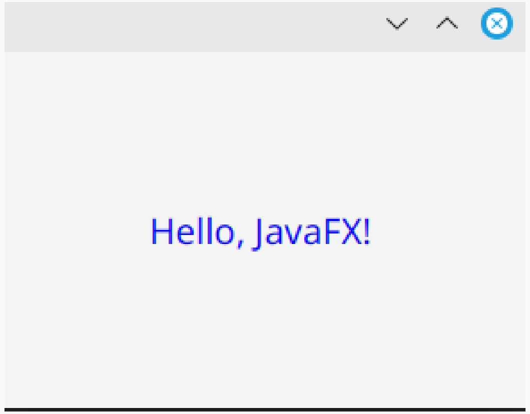
You've seen all of these thousands of times.

Label	Display non-editable text with simple formatting
Text	Display text with advanced styling options
Button	Interactive control for user actions
TextField	Single-line text input for short user entries
TextArea	Multi-line text input for longer content
CheckBox	Multiple independent selections (zero, one, or many)
RadioButton	Mutually exclusive choices (exactly one selection)
ProgressBar	Visual feedback for task progress

Labels and Text

Label is the simplest way to display *non-editable* text:

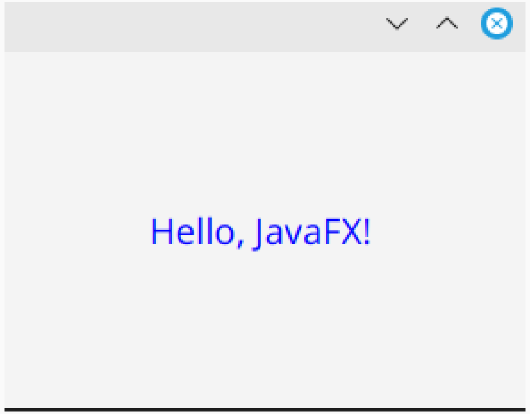
```
Label label = new Label("Hello, JavaFX!");
label.setFont(new Font("Arial", 20));
label.setTextFill(Color.BLUE);
```



Labels and Text

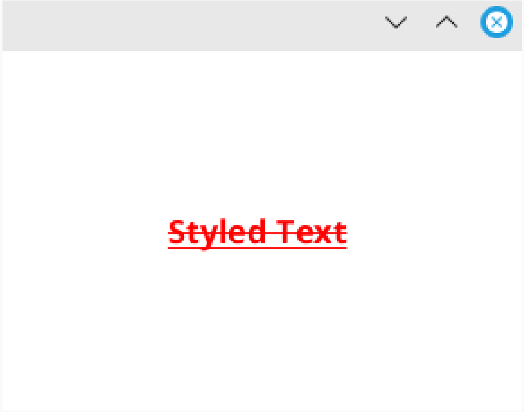
Label is the simplest way to display *non-editable* text:

```
Label label = new Label("Hello, JavaFX!");
label.setFont(new Font("Arial", 20));
label.setTextFill(Color.BLUE);
```



Text provides more control over text rendering:

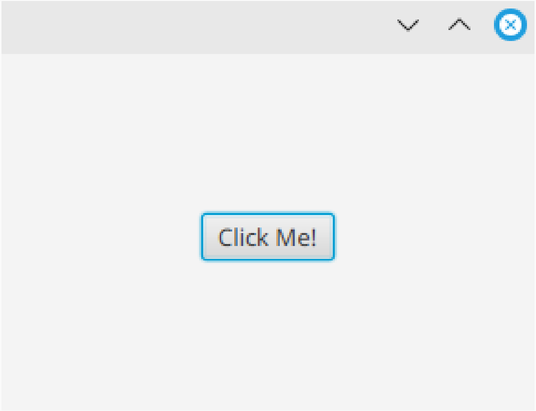
```
Text text = new Text("Styled Text");
text.setFont(Font.font("Verdana", FontWeight.BOLD,
18));
text.setFill(Color.RED);
text.setStrikethrough(true);
text.setUnderline(true);
```



Buttons and Event Handling

Button is the most common interactive control:

```
Button button = new Button("Click Me!");

// Handle click with lambda expression
button.setOnAction(event -> {
    System.out.println("Button was clicked!");
});
```



Interlude: Lambda Expressions

Lambda expressions

✏️ A **lambda expression** is an *anonymous function* that can be used as a value.

Lambda expressions

✏️ A **lambda expression** is an *anonymous function* that can be used as a value.

Named function

```
int add(int x, int y) {
    return x + y
}
```

```
void show(String s) {
    System.out.println(s)
}
```

Lambda expression

```
(x, y) -> x + y
```

```
s -> System.out.println(s)
```

i Lambda expressions allow us to pass around **actions** just like we pass around `int`s.

Lambda expressions: Etymology

Alonzo Church invented the **lambda calculus**, which uses lambda (λ) to indicate functions.

Lambda expressions: Etymology

Alonzo Church invented the **lambda calculus**, which uses lambda (λ) to indicate functions.

$$(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))(\lambda g.\lambda n.n(\lambda x.(\lambda m.\lambda n.\lambda f.mf(nf))(n)(g((\lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(gf))(\lambda u.x)(\lambda u.u))n)))(\lambda f.\lambda x.fx))$$

Lambda expressions: Etymology

Alonzo Church invented the **lambda calculus**, which uses lambda (λ) to indicate functions.

$$(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))(\lambda g.\lambda n.n(\lambda x.(\lambda m.\lambda n.\lambda f.mf(nf))(n)(g((\lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(gf))(\lambda u.x)(\lambda u.u))n)))(\lambda f.\lambda x.fx))$$

Church used the syntax $\lambda x.e$ to represent a function with input x and body e.

Lambda expressions: Etymology

Alonzo Church invented the **lambda calculus**, which uses lambda (λ) to indicate functions.

$$(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))(\lambda g.\lambda n.n(\lambda x.(\lambda m.\lambda n.\lambda f.mf(nf))(n)(g((\lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(gf))(\lambda u.x)(\lambda u.u))n)))(\lambda f.\lambda x.fx))$$

Church used the syntax $\lambda x.e$ to represent a function with input x and body e.

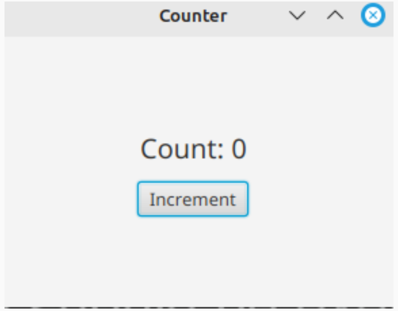
You'll learn much more about this in your programming language course.
For now, you just need to know how to write an *anonymous function* `x -> ...`

Back to GUIs

Example: Label and Button

A simple counter that increments when the button is clicked:

```
public class CounterApp extends Application {
    private int counter = 0;

    public void start(Stage primaryStage) {
        Label label = new Label("Count: 0");
        label.setFont(new Font(20));

        Button button = new Button("Increment");
        button.setOnAction(event -> {
            counter++;
            label.setText("Count: " + counter);
        });

        VBox root = new VBox(10);
        root.setAlignment(Pos.CENTER);
        root.getChildren().addAll(label, button);

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

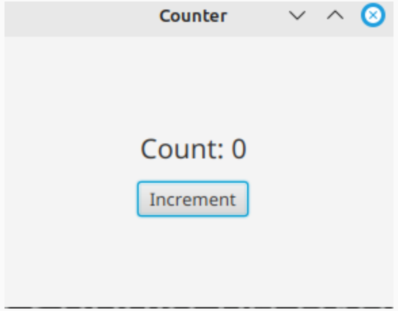


No increments

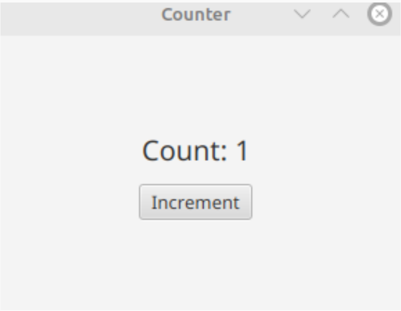
Example: Label and Button

A simple counter that increments when the button is clicked:

```java
public class CounterApp extends Application {
    private int counter = 0;

    public void start(Stage primaryStage) {
        Label label = new Label("Count: 0");
        label.setFont(new Font(20));

        Button button = new Button("Increment");
        button.setOnAction(event -> {
            counter++;
            label.setText("Count: " + counter);
        });

        VBox root = new VBox(10);
        root.setAlignment(Pos.CENTER);
        root.getChildren().addAll(label, button);

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



No increments



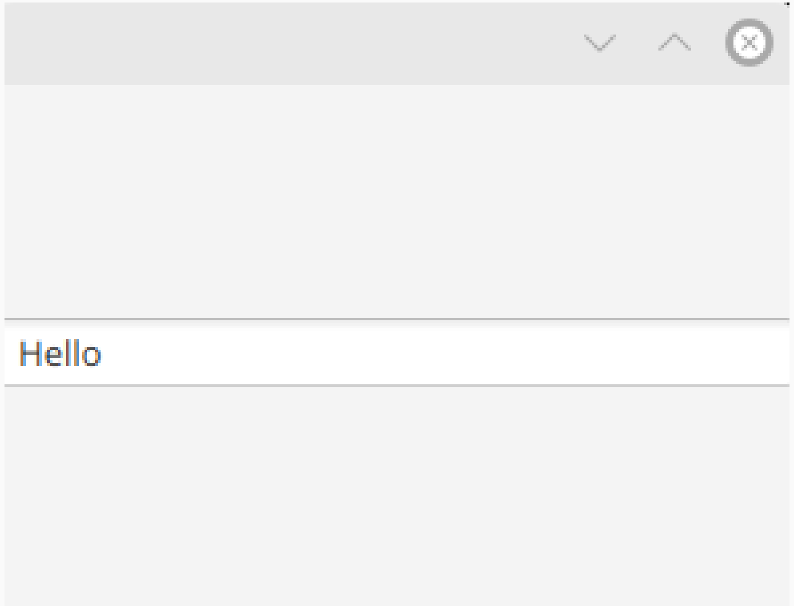
1 increment

TextField for *single-line* text input:

```java
TextField textField = new TextField();
textField.setPromptText("Enter your name");
textField.setPrefColumnCount(20);

// Get the text value
String text = textField.getText();

// Listen for changes
textField.textProperty().addListener((obs, oldText, newText) -> {
    System.out.println("Text changed to: " + newText);
});
```

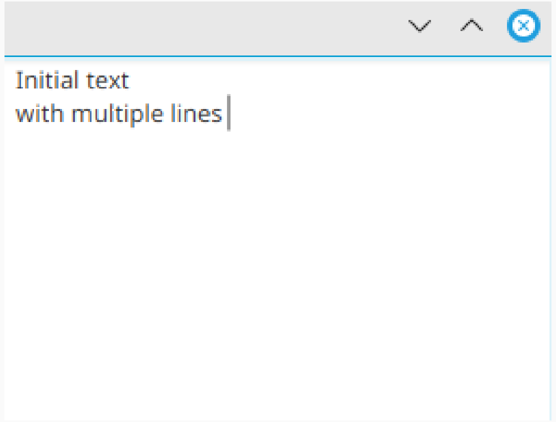
After typing “Hello” in the textfield

```
Text changed to: H
Text changed to: He
Text changed to: Hel
Text changed to: Hell
Text changed to: Hello
```



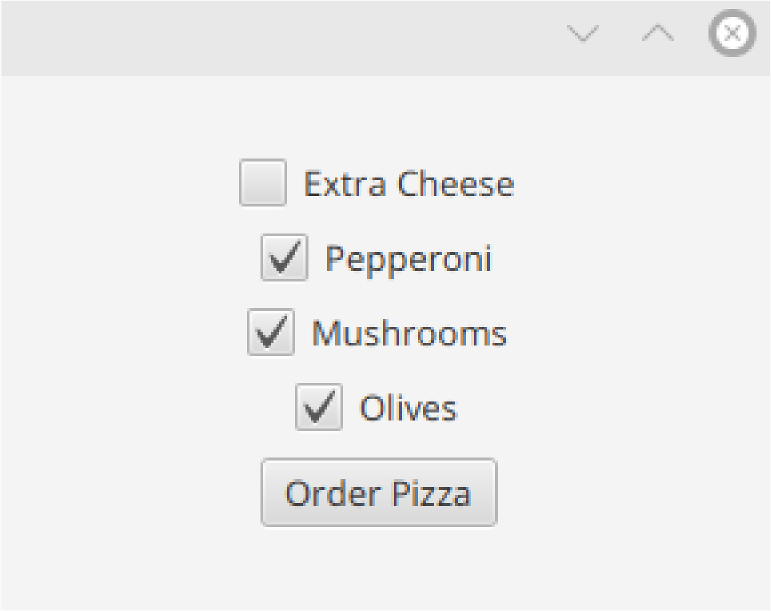
TextArea for *multi-line* text input:

```java
TextArea textArea = new TextArea();
textArea.setPrefRowCount(5);
textArea.setPrefColumnCount(30);
textArea.setWrapText(true);
textArea.setPromptText("Enter comments...");

// Get the text value
String text = textArea.getText();

// Set text programmatically
textArea.setText("Initial text\nwith multiple lines");
```



CheckBox for *multiple independent* choices:

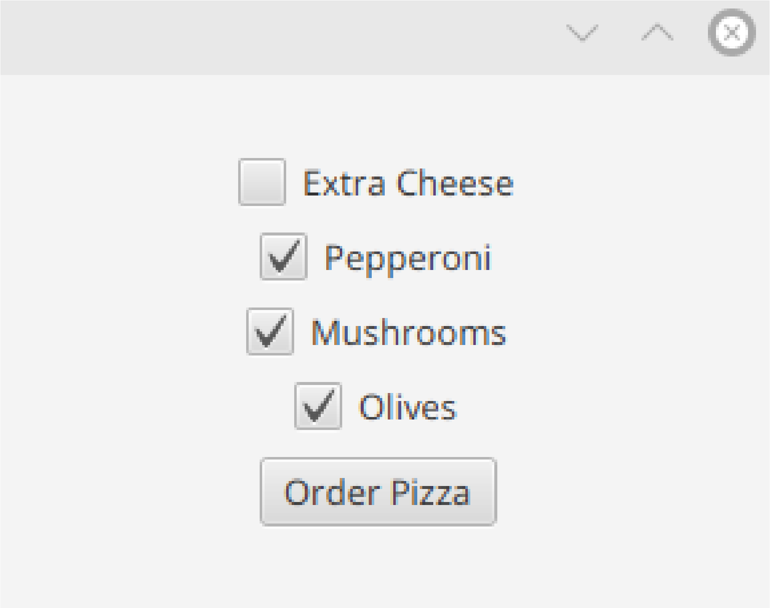
```java
// Create checkboxes
CheckBox cheese = new CheckBox("Extra Cheese");
CheckBox pepperoni = new CheckBox("Pepperoni");
CheckBox mushrooms = new CheckBox("Mushrooms");
CheckBox olives = new CheckBox("Olives");

Button orderBtn = new Button("Order Pizza");
```



,

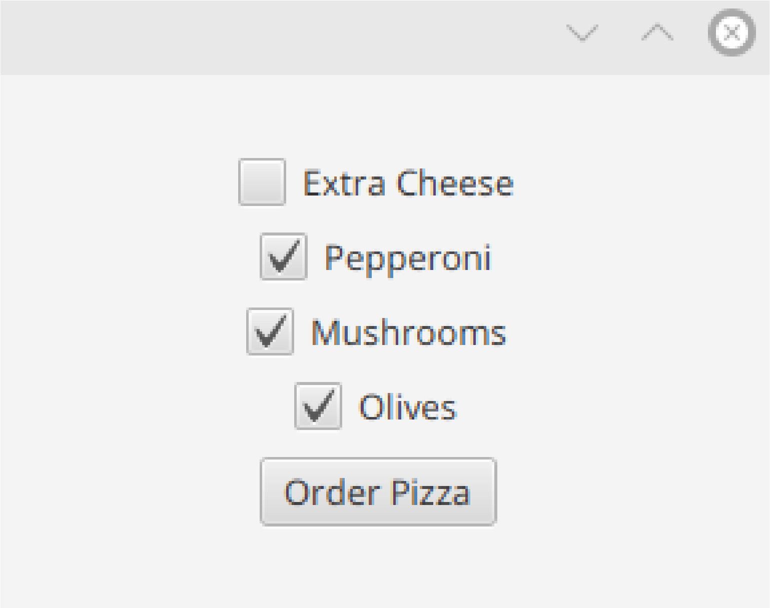
CheckBoxes

```
cheese.setSelected(true);

pepperoni.setOnAction(event -> {
    if (pepperoni.isSelected()) {
        System.out.println("Added pepperoni");
    } else {
        System.out.println("Removed pepperoni");
    }
});

orderBtn.setOnAction(event -> {
    System.out.println("Toppings: ");
    if (cheese.isSelected())
        System.out.println("- Extra Cheese");
    if (pepperoni.isSelected())
        System.out.println("- Pepperoni");
    if (mushrooms.isSelected())
        System.out.println("- Mushrooms");
    if (olives.isSelected())
        System.out.println("- Olives");
});
```



CheckBoxes

```java
cheese.setSelected(true);

pepperoni.setOnAction(event -> {
    if (pepperoni.isSelected()) {
        System.out.println("Added pepperoni");
    } else {
        System.out.println("Removed pepperoni");
    }
});

orderBtn.setOnAction(event -> {
    System.out.println("Toppings: ");
    if (cheese.isSelected())
        System.out.println("- Extra Cheese");
    if (pepperoni.isSelected())
        System.out.println("- Pepperoni");
    if (mushrooms.isSelected())
        System.out.println("- Mushrooms");
    if (olives.isSelected())
        System.out.println("- Olives");
});
```

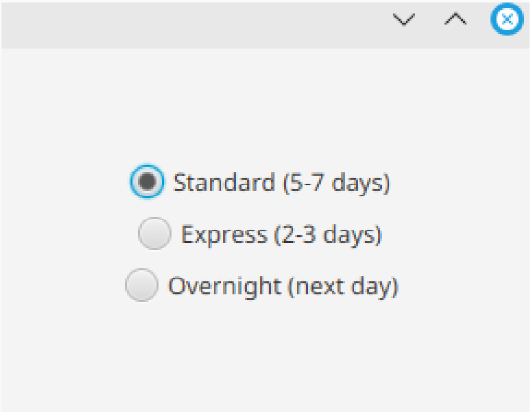


After clicking "Order Pizza":

```
Toppings:
- Pepperoni
- Mushrooms
- Olives
```

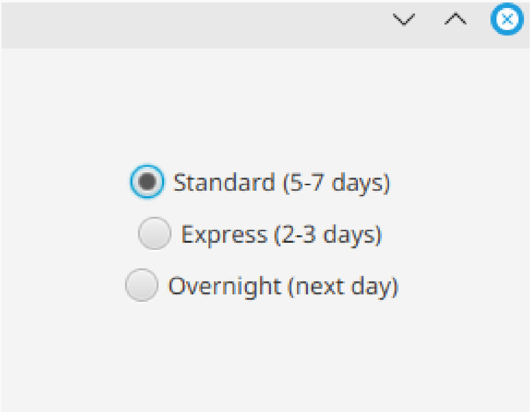
RadioButton for *mutually exclusive* choices:

```
// Create radio buttons for delivery options
RadioButton standard = new RadioButton("Standard (5-7 days)");
RadioButton express = new RadioButton("Express (2-3 days)");
RadioButton overnight = new RadioButton("Overnight (next day)");

// Group them together - only one can be selected
ToggleGroup deliveryGroup = new ToggleGroup();
standard.setToggleGroup(deliveryGroup);
express.setToggleGroup(deliveryGroup);
overnight.setToggleGroup(deliveryGroup);

// Set default selection
standard.setSelected(true);

// Handle selection changes
deliveryGroup.selectedToggleProperty().addListener((obs, old, toggle) -> {
    RadioButton selected = (RadioButton) toggle;
    System.out.println("Delivery: " + selected.getText());
});
```



RadioButton for *mutually exclusive* choices:

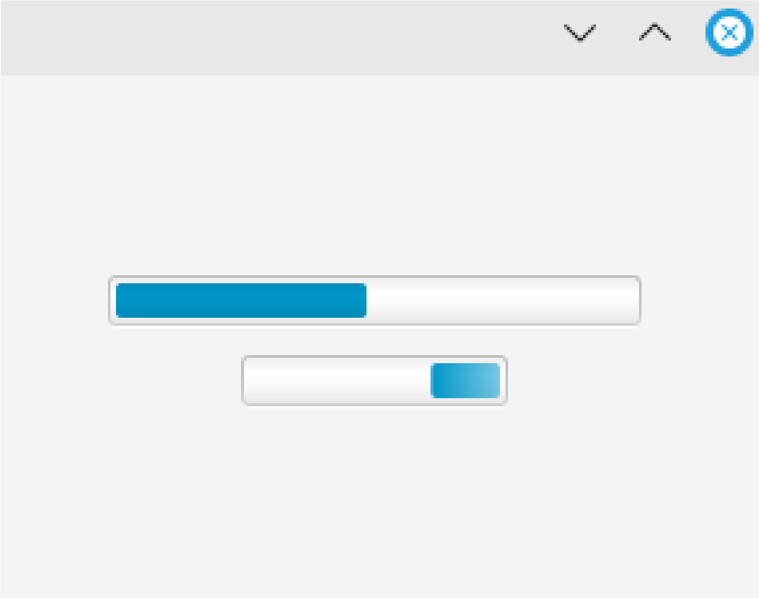
```java
// Create radio buttons for delivery options
RadioButton standard = new RadioButton("Standard (5-7 days)");
RadioButton express = new RadioButton("Express (2-3 days)");
RadioButton overnight = new RadioButton("Overnight (next day)");

// Group them together - only one can be selected
ToggleGroup deliveryGroup = new ToggleGroup();
standard.setToggleGroup(deliveryGroup);
express.setToggleGroup(deliveryGroup);
overnight.setToggleGroup(deliveryGroup);

// Set default selection
standard.setSelected(true);

// Handle selection changes
deliveryGroup.selectedToggleProperty().addListener((obs, old, toggle) -> {
    RadioButton selected = (RadioButton) toggle;
    System.out.println("Delivery: " + selected.getText());
});
```




```
Delivery: Express (2-3 days)
Delivery: Standard (5-7 days)
Delivery: Overnight (next day)
```

ProgressBar shows the progress of a task:

```
// Create progress bar
ProgressBar progressBar = new ProgressBar(0.5);
progressBar.setPrefWidth(200);

// Indeterminate progress (unknown duration)
ProgressBar loadingBar = new ProgressBar();
loadingBar.setProgress(ProgressBar.INDETERMINATE_PROGRESS);
```



First progress bar is set to 50%

Second progress bar moves continuously

Other Important UI Components

JavaFX provides many more UI components for building rich applications:

- **ComboBox**: Dropdown list for selecting from many options
- **ListView**: Display a scrollable list of items
- **TableView**: Display data in rows and columns
- **Slider**: Select numeric values by dragging
- **PasswordField**: TextField that masks input for passwords
- **DatePicker**: Calendar widget for date selection
- **ColorPicker**: Select colors with a visual picker
- **MenuBar**: Traditional application menus (File, Edit, etc.)
- **TabPane**: Organize content into tabs

Layout Management

Panes Overview

Layout panes automatically arrange and position their child nodes:

- **VBox**: Arranges children in a *vertical* column
- **HBox**: Arranges children in a *horizontal* row
- **GridPane**: Arranges children in a flexible *grid* of rows and columns
- **BorderPane**: Divides area into *five regions*: top, bottom, left, right, center
- **StackPane**: *Stacks* children on top of each other (centered by default)
- **FlowPane**: Arranges children in a *flow* that wraps at boundaries

Using panes

TODO 🐑 ML: something about: make the children, make the pane, configure the pane, add the children

VBox arranges children in a *vertical column* from top to bottom:

```java
VBox vbox = new VBox(10);  // 10px space between
vbox.setAlignment(Pos.CENTER);
vbox.setPadding(new Insets(15)); // 15px from edge
vbox.getChildren().addAll(button1, button2, button3);
```

VBox: Vertical Layout

Key properties:

- **Spacing**: The gap between child nodes (in pixels)
- **Alignment**: How children are positioned horizontally within the VBox
  - `Pos.CENTER`, `Pos.TOP_LEFT`, `Pos.TOP_RIGHT`, etc.
- **Padding**: Space between the VBox border and its children
  - `new Insets(15)` adds 15px on all sides
  - `new Insets(10, 20, 10, 20)` for top, right, bottom, left

HBox: Horizontal Layout

HBox arranges children in a *horizontal row* from left to right:

```java
HBox hbox = new HBox(10);  // 10px space between
hbox.setAlignment(Pos.CENTER);
hbox.setPadding(new Insets(15)); // 15px from edge
hbox.getChildren().addAll(button1, button2, button3);
```

GridPane

GridPane arranges children in a flexible *grid* of rows and columns:

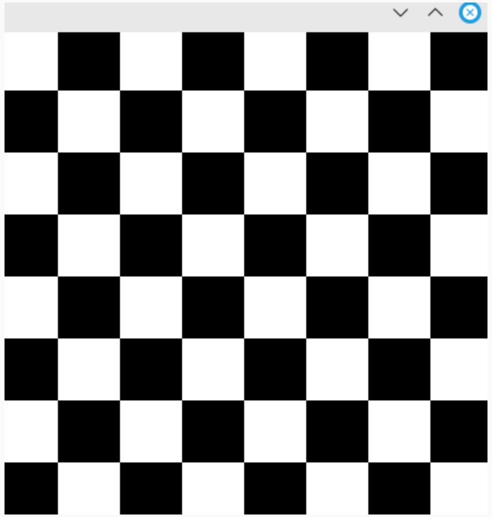
```java
GridPane grid = new GridPane();
grid.setHgap(10);     // Horizontal gap between columns
grid.setVgap(10);     // Vertical gap between rows
grid.setPadding(new Insets(20));

// Add nodes at specific column and row positions
grid.add(button1, 0, 0);      // Column 0, Row 0
grid.add(button2, 1, 0);      // Column 1, Row 0
grid.add(button3, 1, 2);      // Column 1, Row 2
```

Use GridPane for forms and any layout requiring precise row/column placement.

GridPane Example: Checkerboard

Creating an 8×8 checkerboard with alternating colors:

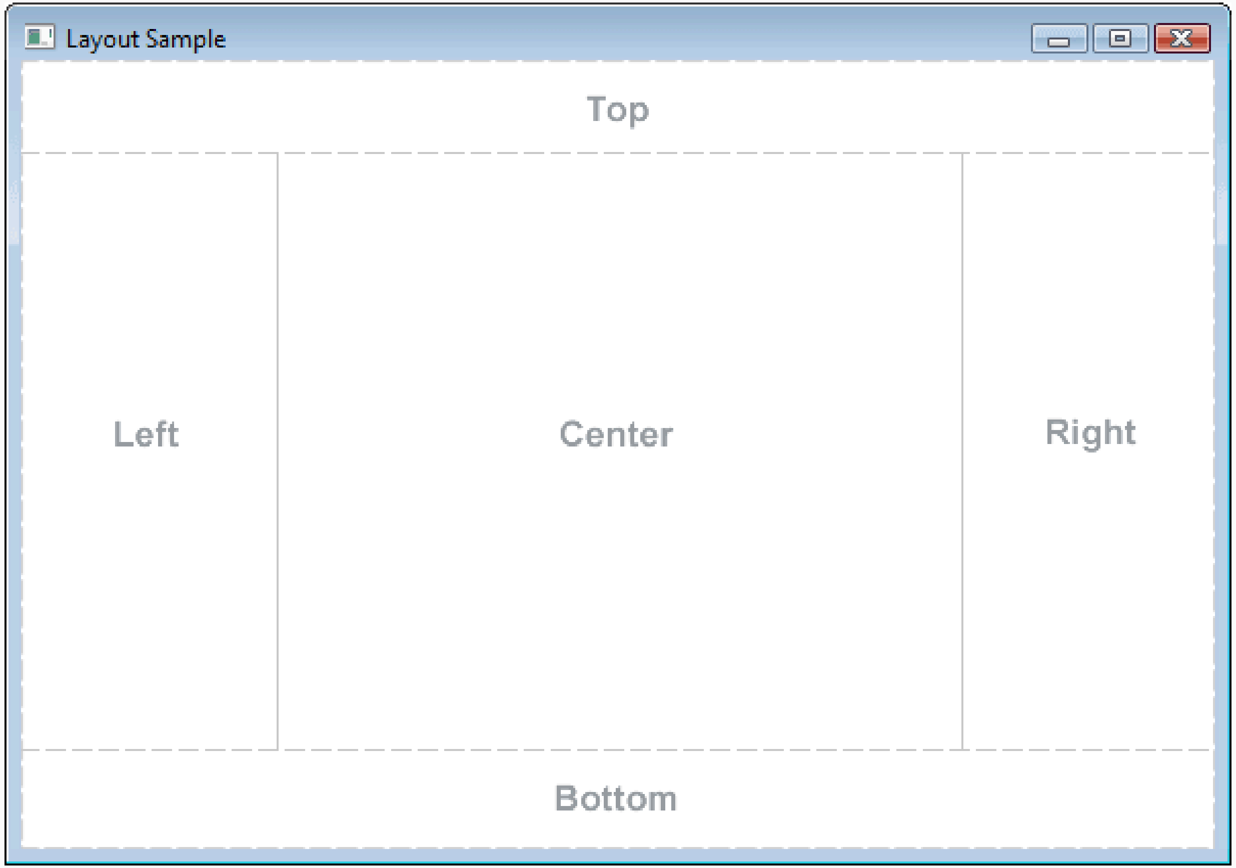
```java
GridPane board = new GridPane();
for (int row = 0; row < 8; row++) {
    for (int col = 0; col < 8; col++) {
        Rectangle square = new Rectangle(50, 50);

        // Alternate colors based on row and column
        if ((row + col) % 2 == 0) {
            square.setFill(Color.WHITE);
        } else {
            square.setFill(Color.BLACK);
        }

        board.add(square, col, row);
    }
}
```



GridPane makes it easy to create grid-based layouts like game boards.

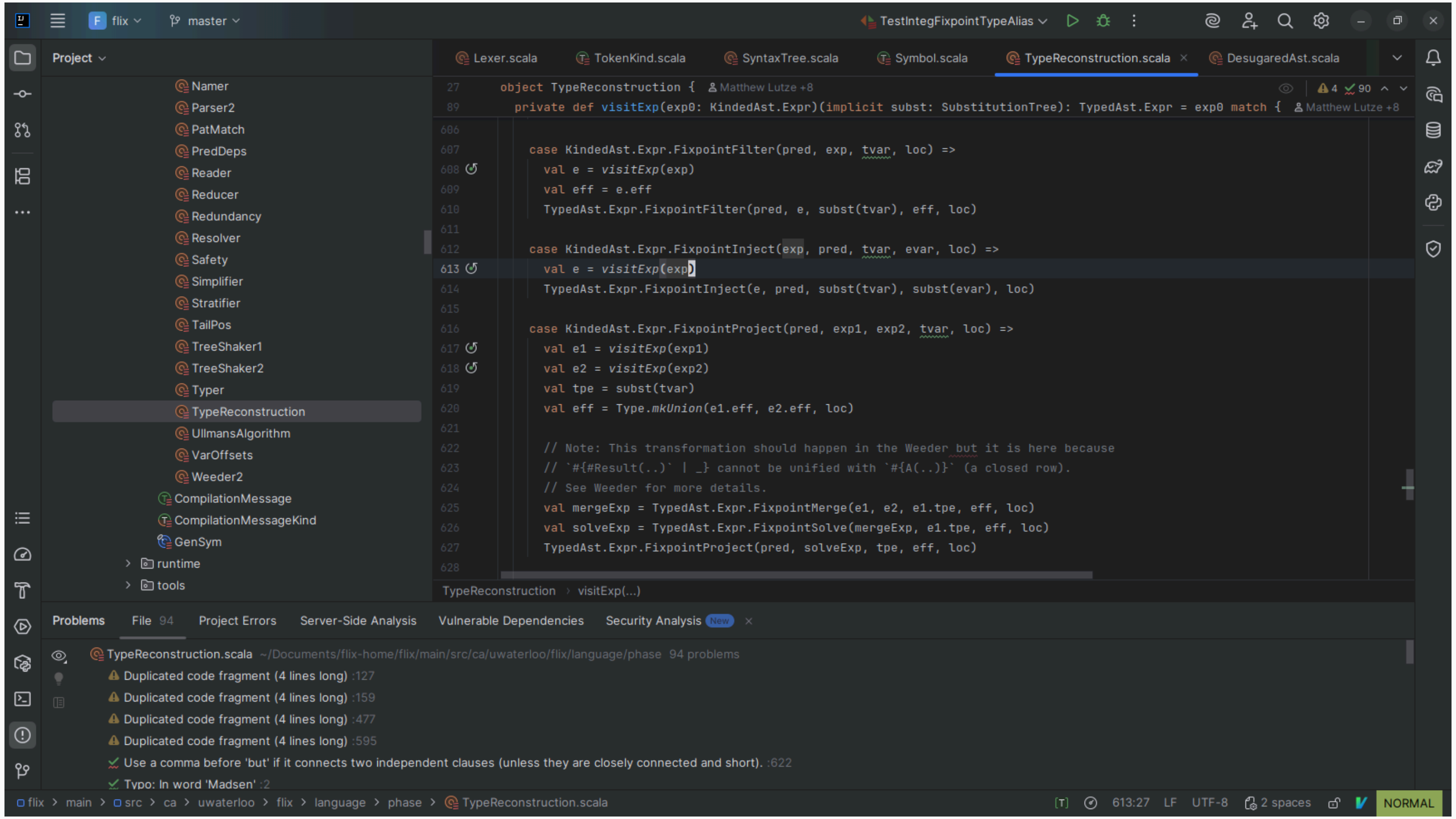
BorderPane

BorderPane divides the layout into *five regions*:



BorderPane

IntelliJ uses **BorderPane** for its interface:

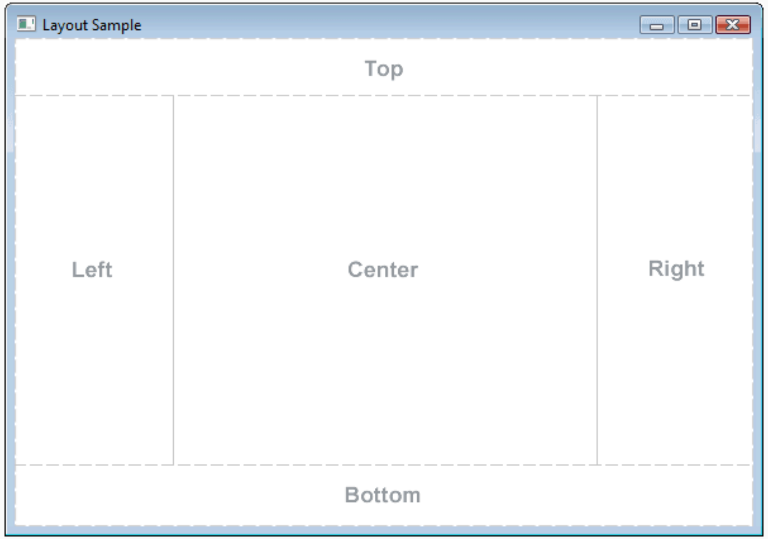


BorderPane

Regions:

- **Top** and **Bottom**: Typically for menus and status bars
- **Left** and **Right**: Navigation or tool panels
- **Center**: Main content area (expands to fill available space)

```
BorderPane border = new BorderPane();

// Set content for each region
border.setTop(new MenuBar());
border.setBottom(new Label("Status Bar"));
border.setLeft(new VBox(navigationButtons));
border.setRight(new VBox(toolButtons));
border.setCenter(new TextArea("Main Content"));
```



Use BorderPane for traditional application layouts with toolbars and sidebars.

Combining Layouts

Nest panes inside each other to create sophisticated layouts:

- Use **BorderPane** as the *root container* for traditional app layouts
- Place **VBox** or **HBox** inside regions for *linear arrangements*
- Add **GridPane** in the center for *forms and data entry*
- Embed **StackPane** for *overlaying elements* like notifications
- Combine **ScrollPane** with any pane when content might *overflow*

Combining Layouts

Nest panes inside each other to create sophisticated layouts:

- Use **BorderPane** as the *root container* for traditional app layouts
- Place **VBox** or **HBox** inside regions for *linear arrangements*
- Add **GridPane** in the center for *forms and data entry*
- Embed **StackPane** for *overlaying elements* like notifications
- Combine **ScrollPane** with any pane when content might *overflow*

Benefits of combining layouts:

- Each pane handles its specific layout task well
- Separation of concerns makes code more maintainable
- Easier to modify individual sections without affecting others

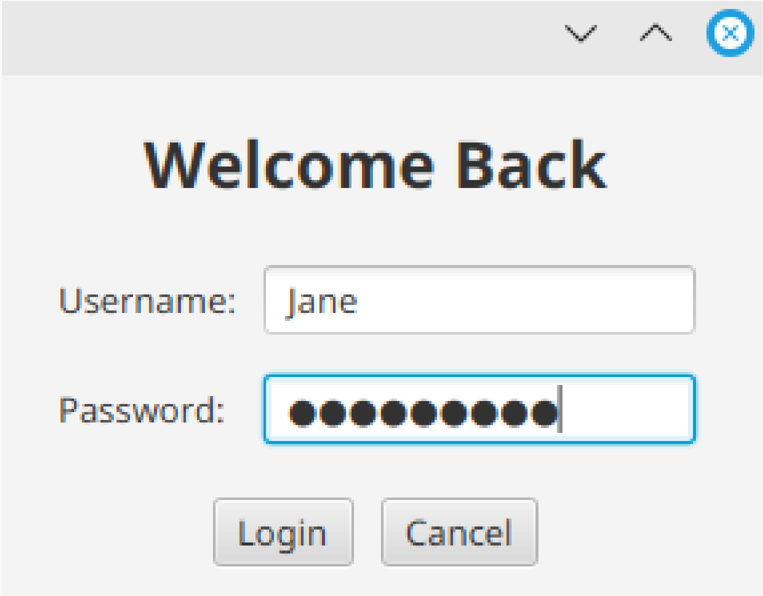
Combining Layouts: Example

Creating a login form with nested panes:

```java
// Main container: VBox
VBox root = new VBox(20);
root.setPadding(new Insets(30));
root.setAlignment(Pos.CENTER);

// Title
Label title = new Label("Welcome Back");
title.setFont(Font.font("Arial",
    FontWeight.BOLD, 24));

// Form: GridPane for layout
GridPane form = new GridPane();
form.setHgap(10);
form.setVgap(15);
form.setAlignment(Pos.CENTER);
```

```java
// Form fields
Label userLabel = new Label("Username:");
TextField userField = new TextField();
Label passLabel = new Label("Password:");
PasswordField passField = new PasswordField();

form.add(userLabel, 0, 0);
form.add(userField, 1, 0);
form.add(passLabel, 0, 1);
form.add(passField, 1, 1);

// Buttons: HBox for horizontal layout
HBox buttons = new HBox(10);
buttons.setAlignment(Pos.CENTER);
Button loginBtn = new Button("Login");
Button cancelBtn = new Button("Cancel");
buttons.getChildren().addAll(loginBtn, cancelBtn);

root.getChildren().addAll(title, form, buttons);
```

Combining Layouts: Example



FlowPane

FlowPane arranges children in a *flow* that wraps at boundaries:

```
FlowPane flow = new FlowPane();
flow.setHgap(10);  // Horizontal gap between nodes
flow.setVgap(10);  // Vertical gap between rows
flow.setPrefWrapLength(300);  // Preferred width before
wrapping

// Add multiple buttons - they will wrap automatically
for (int i = 1; i <= 12; i++) {
    Button btn = new Button("Button " + i);
    flow.getChildren().add(btn);
}
```

FlowPane

FlowPane arranges children in a *flow* that wraps at boundaries:

```
FlowPane flow = new FlowPane();
flow.setHgap(10);  // Horizontal gap between nodes
flow.setVgap(10);  // Vertical gap between rows
flow.setPrefWrapLength(300);  // Preferred width before
wrapping

// Add multiple buttons - they will wrap automatically
for (int i = 1; i <= 12; i++) {
    Button btn = new Button("Button " + i);
    flow.getChildren().add(btn);
}
```

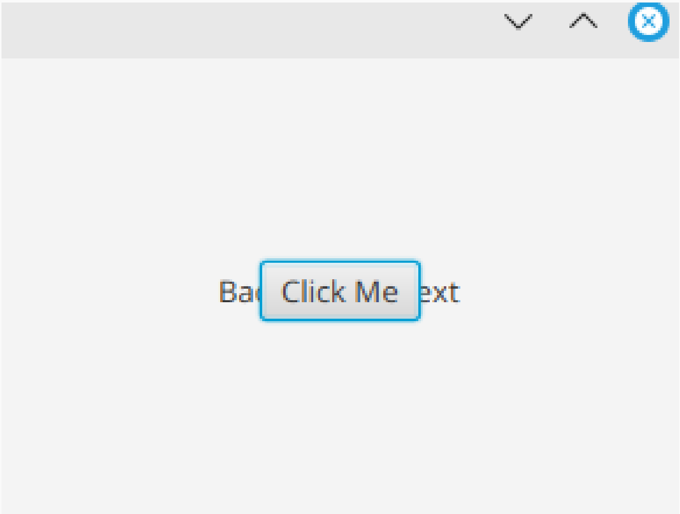
Key features:

- Children flow from *left to right* (or top to bottom with vertical orientation)
- Automatically *wraps* to next row when space runs out
- Resizes dynamically with window

StackPane

StackPane layers children on *top of each other* (centered by default):

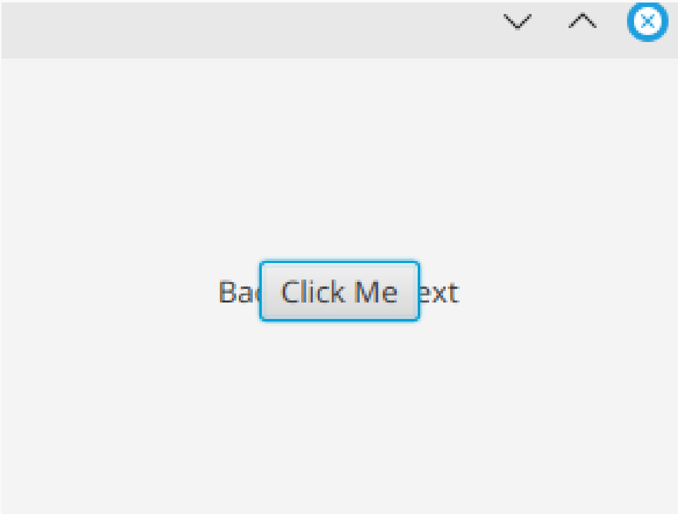
```java
StackPane stack = new StackPane();

Button button = new Button("Click Me");
Label label = new Label("Background Text");

// Children are stacked in order added
stack.getChildren().addAll(label, button);
// label behind, button on top
```



StackPane

StackPane layers children on *top of each other* (centered by default):

```java
StackPane stack = new StackPane();

Button button = new Button("Click Me");
Label label = new Label("Background Text");

// Children are stacked in order added
stack.getChildren().addAll(label, button);
// label behind, button on top
```



Key concepts:

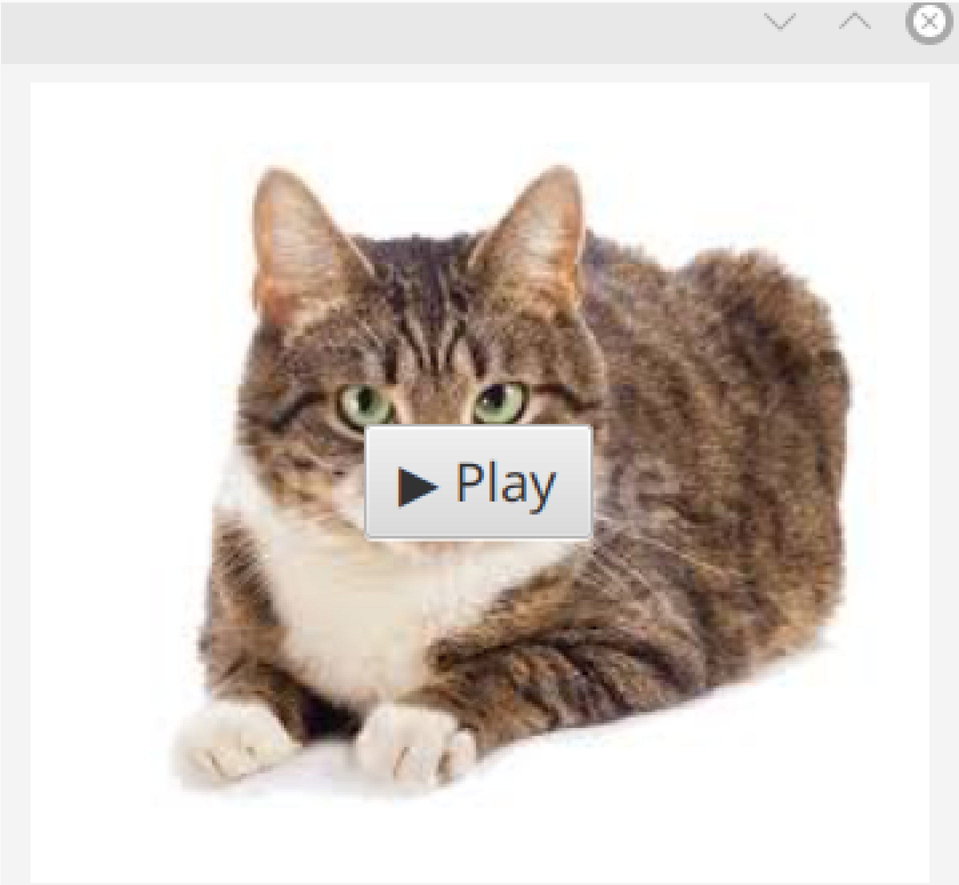
- Children are *centered* by default
- Order matters: first added = *bottom layer*, last added = *top layer*
- All children occupy the *same space* in the center

Use StackPane for overlaying elements like watermarks, loading indicators, or dialogs.

Example: StackPane

Placing an interactive button on top of an image:

```
StackPane stack = new StackPane();

// Load and display an image
ImageView imageView = new
ImageView("background.jpg");
imageView.setFitWidth(400);
imageView.setFitHeight(300);
imageView.setPreserveRatio(true);

// Create a play button to overlay on the image
Button playButton = new Button("▶ Play");
playButton.setStyle("-fx-font-size: 20px;");

// Stack them: image at bottom, button on top
stack.getChildren().addAll(imageView, playButton);
stack.setAlignment(Pos.CENTER);
```



Q:

What is wrong here?

```java
public void start(Stage stage) {
    Button submitButton = new Button("Submit");

    // Left panel
    VBox leftPanel = new VBox(10);
    leftPanel.getChildren().addAll(new Label("Form"), submitButton);

    // Right panel
    VBox rightPanel = new VBox(10);
    rightPanel.getChildren().addAll(new Label("Preview"), submitButton);

    HBox root = new HBox(20, leftPanel, rightPanel);
    Scene scene = new Scene(root, 400, 300);
    stage.setScene(scene);
    stage.show();
}
```

Summary of Layout Management

Choose the right pane for your needs:

- **VBox/HBox**: Simple linear layouts (vertical/horizontal)
- **GridPane**: Forms and grid-based layouts with rows and columns
- **BorderPane**: Traditional app layout with toolbar, sidebar, and content areas
- **StackPane**: Overlay elements on top of each other
- **FlowPane**: Responsive layouts that wrap content automatically

Summary of Layout Management

Choose the right pane for your needs:

- **VBox/HBox**: Simple linear layouts (vertical/horizontal)
- **GridPane**: Forms and grid-based layouts with rows and columns
- **BorderPane**: Traditional app layout with toolbar, sidebar, and content areas
- **StackPane**: Overlay elements on top of each other
- **FlowPane**: Responsive layouts that wrap content automatically

Other useful panes:

- **TilePane**: Uniform grid where all cells have the same size
- **ScrollPane**: Add scrollbars to any content that exceeds available space

Q:

1. Which layout pane would you use to create a toolbar with buttons aligned horizontally at the top of your application?
2. You need to create a login form with labels and text fields aligned in two columns. Which layout pane is most appropriate?
3. How would you create a photo viewer where control buttons appear on top of the image when the user hovers over it?

Q:

What happens when you run this program?

```java
public class MyApp extends Application {
    public void start(Stage primaryStage) {
        VBox layout = new VBox(10);

        Label label = new Label("Welcome!");
        Button button = new Button("Click Me");

        layout.getChildren().add(label);

        Scene scene = new Scene(layout, 300, 200);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

Event Handling

What are Events?

We can now design simple interfaces in JavaFX with components and layouts.

But in general, we have not defined any interactions.

E.g. what happens when we press a button?

Events are notifications when something has happened.

A Brief Introduction to Lambdas

Lambda expressions are *anonymous functions* that simplify event handling:

- A lambda is a *short way to write a function* without declaring a separate method.
- Perfect for *single-use functions* like event handlers.
- Replaces verbose anonymous inner classes with *concise syntax*.
- JavaFX uses lambdas extensively for *handling user interactions*.

Lambda expressions are *anonymous functions* that simplify event handling:

- A lambda is a *short way to write a function* without declaring a separate method.
- Perfect for *single-use functions* like event handlers.
- Replaces verbose anonymous inner classes with *concise syntax*.
- JavaFX uses lambdas extensively for *handling user interactions*.

Lambda syntax:

- `(parameters) -> expression`
- `(parameters) -> { statements; }`

Examples: Using Lambdas with JavaFX

Handle button clicks with a simple lambda expression:

```java
button.setOnAction(event -> System.out.println("Clicked!"));
```

Examples: Using Lambdas with JavaFX

Handle button clicks with a simple lambda expression:

```
button.setOnAction(event -> System.out.println("Clicked!"));
```

Listen for text changes and print them to the console:

```
textField.textProperty().addListener((obs, oldText, newText) ->
    System.out.println("New text: " + newText)
);
```

Examples: Using Lambdas with JavaFX

Handle button clicks with a simple lambda expression:

```java
button.setOnAction(event -> System.out.println("Clicked!"));
```

Listen for text changes and print them to the console:

```java
textField.textProperty().addListener((obs, oldText, newText) ->
    System.out.println("New text: " + newText)
);
```

React to mouse hover events with console output:

```java
node.setOnMouseEntered(e -> System.out.println("Mouse entered"));
node.setOnMouseExited(e -> System.out.println("Mouse exited"));
```

Captured Variables (1/2)

Lambdas can access variables from their surrounding scope, but with restrictions:

- Variables must be **effectively final** (never reassigned)
- This prevents issues with concurrent access and scope confusion

Example - This works:

```
String prefix = "User clicked: ";  // effectively final
button.setOnAction(event -> {
    System.out.println(prefix + "button");  // OK - can access prefix
});
```

Captured Variables (2/2)

Example - This doesn't work:

```java
int counter = 0;
button.setOnAction(event -> {
    counter++;  // Error: counter is not effectively final
    System.out.println("Count: " + counter);
});
```

Use instance variables or local final variables to share data with lambdas

Event Types and Sources

Common event types in JavaFX:

- `ActionEvent` occurs when buttons are clicked or Enter is pressed in text fields.
- `MouseEvent` captures mouse clicks, movement, and hovering.
- `KeyEvent` detects key presses and releases on the keyboard.
- `WindowEvent` tracks window operations like opening, closing, or resizing.

Event Types and Sources

Common event types in JavaFX:

- `ActionEvent` occurs when buttons are clicked or Enter is pressed in text fields.
- `MouseEvent` captures mouse clicks, movement, and hovering.
- `KeyEvent` detects key presses and releases on the keyboard.
- `WindowEvent` tracks window operations like opening, closing, or resizing.

Event sources generate specific events:

- A `Button` generates `ActionEvent` when clicked.
- A `TextField` generates both `ActionEvent` and `KeyEvent`.
- Any `Node` can generate `MouseEvent` and `KeyEvent`.
- A `Stage` generates `WindowEvent` for window operations.

Each event contains details about what happened, such as mouse position or which key was pressed.

Action Events

`ActionEvent` is the most common event for user interactions.

Common sources of ActionEvents:

- `Button` - when clicked
- `TextField` - when Enter is pressed
- `MenuItem` - when selected from a menu
- `Hyperlink` - when clicked

Key characteristics:

- Represents a *completed action* by the user
- Most UI controls support `setOnAction` method
- Simple to handle with lambda expressions

Examples: Action Events

Handle a button click:

```java
button.setOnAction(event -> {
    System.out.println("Button clicked!");
});
```

Examples: Action Events

Handle a button click:

```
button.setOnAction(event -> {
    System.out.println("Button clicked!");
});
```

Process text field submission when Enter is pressed:

```
textField.setOnAction(event -> {
    String text = textField.getText();
    System.out.println("Submitted: " + text);
    textField.clear();
});
```

Keyboard Events

`KeyEvent` lets you respond to keyboard input:

Three types of keyboard events:

- `setOnKeyPressed` - when a key is pressed down
- `setOnKeyReleased` - when a key is released
- `setOnKeyTyped` - when a character is typed

Keyboard Events

`KeyEvent` lets you respond to keyboard input:

Three types of keyboard events:

- `setOnKeyPressed` - when a key is pressed down
- `setOnKeyReleased` - when a key is released
- `setOnKeyTyped` - when a character is typed

Example: Detecting specific keys:

```
scene.setOnKeyPressed(event -> {
    if (event.getCode() == KeyCode.ENTER) {
        System.out.println("Enter pressed!");
    } else if (event.getCode() == KeyCode.ESCAPE) {
        System.out.println("Escape pressed!");
    } else if (event.isControlDown() && event.getCode() == KeyCode.S) {
        System.out.println("Ctrl+S pressed - Save!");
    }
});
```

Mouse Events

`MouseEvent` captures various mouse interactions:

Common mouse event handlers:

- `setOnMouseClicked` - mouse button clicked and released
- `setOnMousePressed` - mouse button pressed down
- `setOnMouseReleased` - mouse button released
- `setOnMouseEntered` - mouse cursor enters a node
- `setOnMouseExited` - mouse cursor leaves a node
- `setOnMouseMoved` - mouse moves within a node

Mouse Events

`MouseEvent` captures various mouse interactions:

Common mouse event handlers:

- `setOnMouseClicked` - mouse button clicked and released
- `setOnMousePressed` - mouse button pressed down
- `setOnMouseReleased` - mouse button released
- `setOnMouseEntered` - mouse cursor enters a node
- `setOnMouseExited` - mouse cursor leaves a node
- `setOnMouseMoved` - mouse moves within a node

Example: Creating interactive buttons:

```
button.setOnMouseEntered(e -> button.setTextFill(Color.BLUE));
button.setOnMouseExited(e -> button.setTextFill(Color.BLACK));
button.setOnMousePressed(e -> button.setTextFill(Color.RED));
button.setOnMouseReleased(e -> button.setTextFill(Color.BLUE));
```

Declarative Layouts

UI as Code

Writing user interfaces entirely in code can become **tedious** for complex layouts:

- Lots of repetitive object creation and property setting
- Hard to visualize the final layout while writing code
- Difficult for designers to collaborate without programming knowledge
- Makes quick design iterations and prototyping slower
- Code becomes verbose and harder to maintain for large UIs

Declarative markup languages like FXML solve these problems

What is FXML?

FXML is an XML-based markup language for defining JavaFX user interfaces:

- Separates UI design from application logic
- Makes layouts more readable and maintainable
- Supports visual design tools like Scene Builder
- Allows designers and developers to work independently

What is FXML?

FXML is an XML-based markup language for defining JavaFX user interfaces:

- Separates UI design from application logic
- Makes layouts more readable and maintainable
- Supports visual design tools like Scene Builder
- Allows designers and developers to work independently

Key benefits:

- Complex layouts are easier to understand in XML format
- UI changes don't require recompiling Java code
- Better organization of large applications
- Supports internationalization and styling

Java vs. FXML

Java Code:

```java
VBox root = new VBox(20);
root.setPadding(new Insets(30));
root.setAlignment(Pos.CENTER);

Label title = new Label("Login");
title.setFont(Font.font(24));

GridPane form = new GridPane();
form.setHgap(10);
form.setVgap(15);

Label userLabel = new Label("Username:");
TextField userField = new TextField();
Label passLabel = new Label("Password:");
PasswordField passField = new PasswordField();

form.add(userLabel, 0, 0);
form.add(userField, 1, 0);
form.add(passLabel, 0, 1);
form.add(passField, 1, 1);

Button loginBtn = new Button("Login");
root.getChildren().addAll(title, form, loginBtn);
```

FXML:

```xml
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<?import javafx.geometry.Insets?>

<VBox spacing="20" alignment="CENTER">
    <padding>
        <Insets top="30" right="30" bottom="30" left="30"/>
    </padding>

    <Label text="Login" style="-fx-font-size: 24px"/>

    <GridPane hgap="10" vgap="15">
        <Label text="Username:" GridPane.columnIndex="0" GridPane.rowIndex="0"/>
        <TextField GridPane.columnIndex="1" GridPane.rowIndex="0"/>
        <Label text="Password:" GridPane.columnIndex="0" GridPane.rowIndex="1"/>
        <PasswordField GridPane.columnIndex="1" GridPane.rowIndex="1"/>
    </GridPane>

    <Button text="Login"/>
</VBox>
```

Declarative UI: A Common Pattern

Similar declarative approaches exist in other frameworks:

React with Chakra UI (Web):

```
<VStack spacing={5} align="center" padding={8}>
    <Heading size="lg">Login</Heading>
    <SimpleGrid columns={2} spacing={3}>
        <FormLabel>Username:</FormLabel>
        <Input />
        <FormLabel>Password:</FormLabel>
        <Input type="password" />
    </SimpleGrid>
    <Button colorScheme="blue">Login</Button>
</VStack>
```

WPF XAML (Windows):

```
<StackPanel Orientation="Vertical" HorizontalAlignment="Center">
    <TextBlock Text="Login" FontSize="24" HorizontalAlignment="Center"/>
    <Grid>
        <TextBlock Text="Username:" Grid.Row="0" Grid.Column="0"/>
        <TextBox Grid.Row="0" Grid.Column="1"/>
        <TextBlock Text="Password:" Grid.Row="1" Grid.Column="0"/>
        <PasswordBox Grid.Row="1" Grid.Column="1"/>
    </Grid>
    <Button Content="Login"/>
</StackPanel>
```

Declarative UI markup is popular across different platforms and frameworks.

Compiler Error of the Week

TODO

Live Programming

- TODO

Sources for images and slides

- https://introcs.cs.princeton.edu/java/lectures/
- https://en.wikipedia.org/wiki/Graphical_user_interface#/media/File:Example_of_a_GUI.png
- https://en.wikipedia.org/wiki/Graphical_user_interface#/media/File:Example_of_a_GUI.png
- https://www.harmonyanimalhospital.net/wp-content/uploads/2018/08/senior-cat-stage.png