



Graphical Interface and Application(I3305)

Chapter 4: JavaFX



Lebanese University
Faculty of Science 1 - Department of Computer Science

Abed EL Safadi

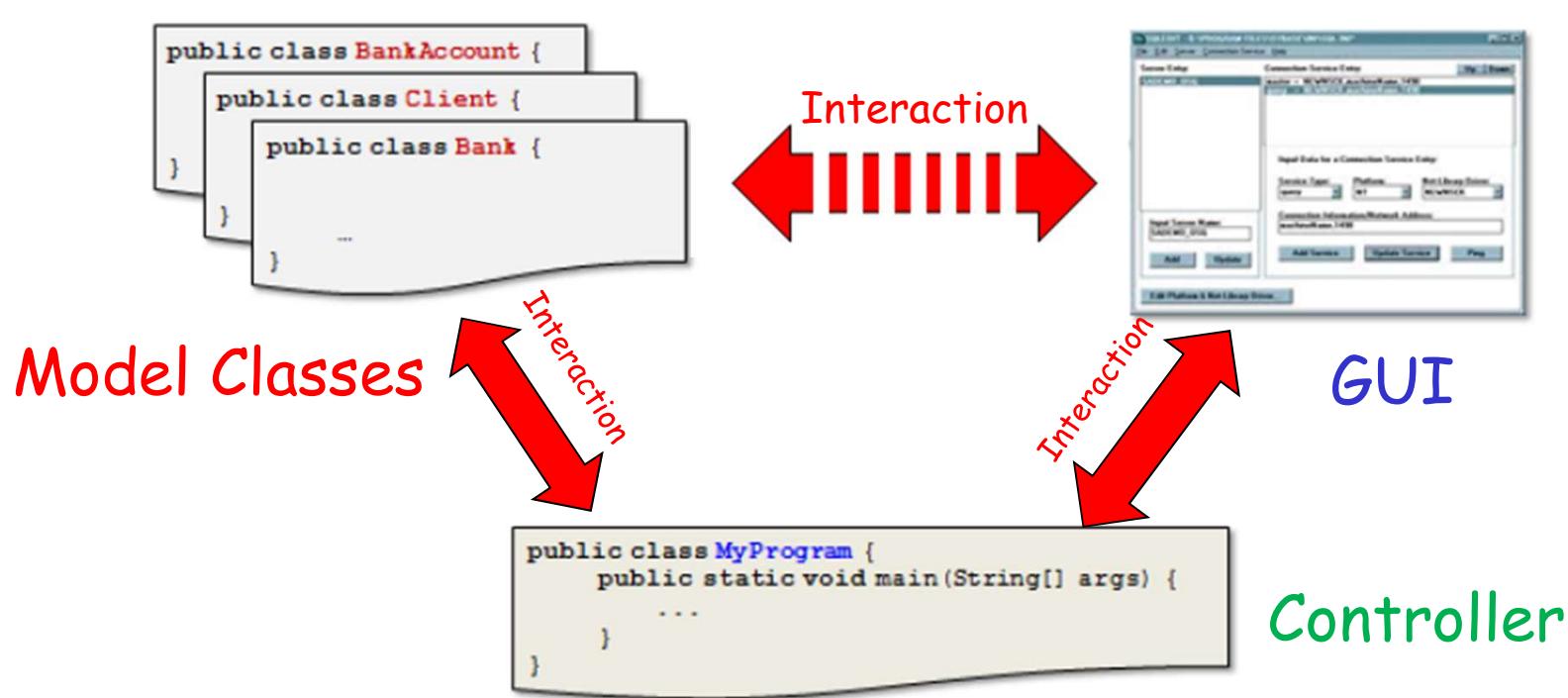
What is a GUI?

A way to communicate what you want to a computer application (or computer operating system) using graphical symbols, rather than typing the instructions in

Ex: We may interact with our bank account electronically by using a web browser, or phone app or dedicated stand-alone software from the bank



MVC in JavaFx GUI



The **model** of an application consists of all classes that represent the "business logic" part of the application

A **GUI** is a user interface that makes use of one or more windows to interact with the user.

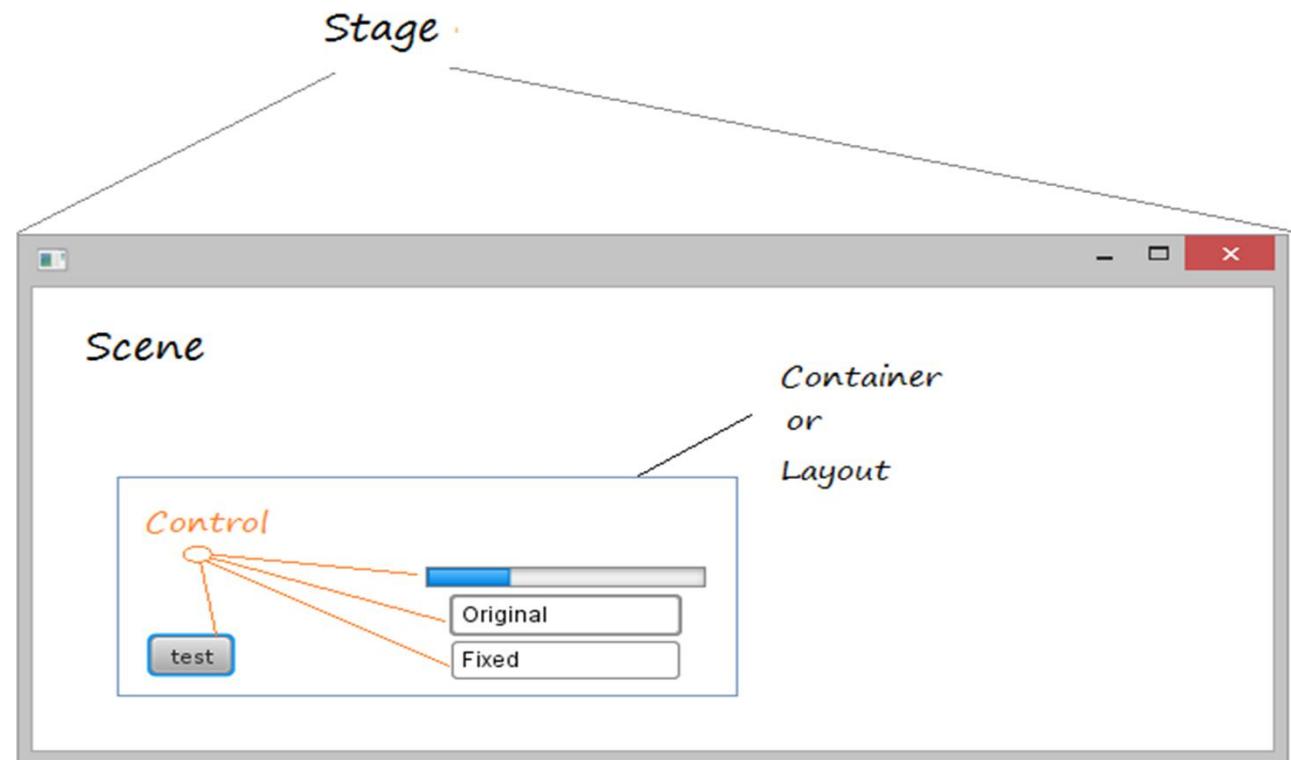
The **controller** joins the model with the GUI

What is JavaFX?

- **JavaFX** is a new framework for developing Java GUI (Graphical User Interface) programs.
- **Swing** and **AWT** are replaced by the JavaFX platform for developing rich Internet applications.
- GUI programs will include visual components such as: buttons, areas for typing in text, drop-down menus for selecting options, areas for displaying results, and so on. Those are known as **User Interface Controls (UI Controls)**.

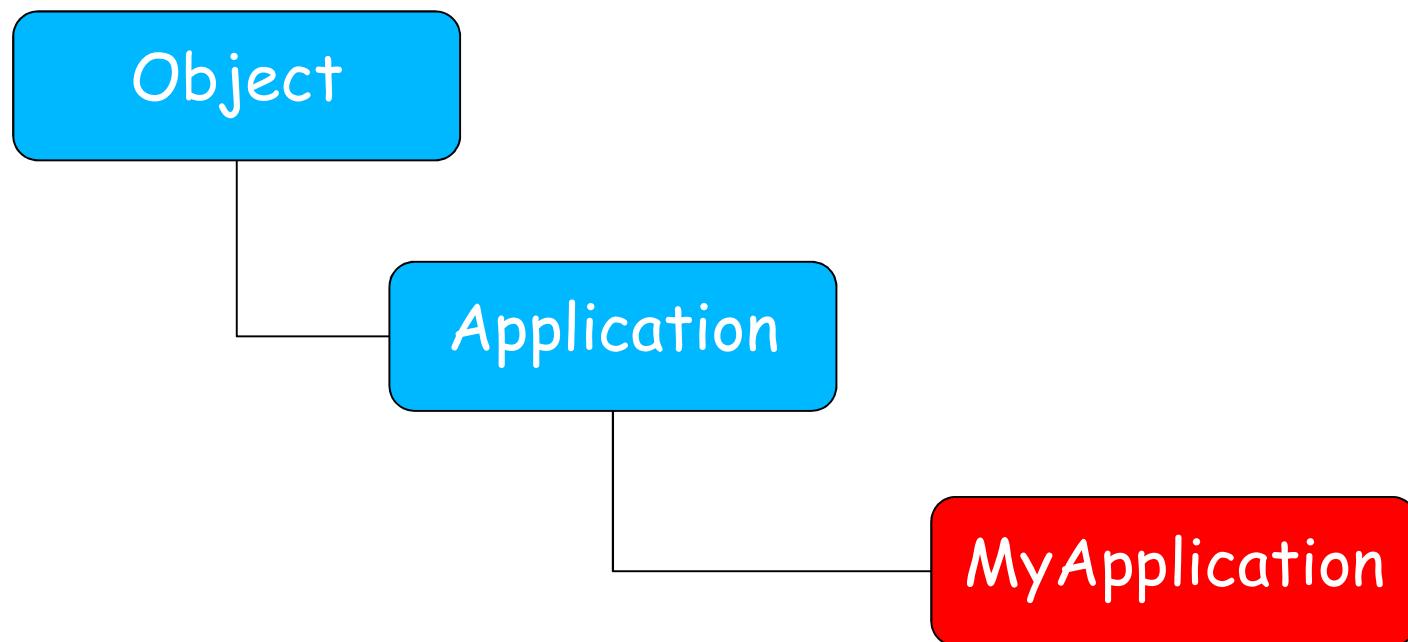
Basic Structure of JavaFX

- Application
- Override the start(Stage) method
- Stage, Scene, and Nodes



Continue: Basic Structure of JavaFX

When using JavaFX, all of our main windows for our applications will be instances of the **Application** class (i.e., we will **extend** that class) which is in the `javafx` package.



Continue: Basic Structure of JavaFX

To *launch* (i.e., start) the application, we call a **launch()** method from our main method. Upon startup, the application will call a **start()** method ... which you must write.

```
import javafx.application.Application;
import javafx.stage.Stage;

public class MyApplication extends Application {
    // Called automatically to start the application (you must write this)
    public void start(Stage primaryStage) { ... }

    public static void main(String[] args) {
        launch(args);    // Get everything going (call this exactly once)
    }
}
```

Here is a template of the bare minimum that you need to start a JavaFX application, although this code does not do anything.

Continue: Basic Structure of JavaFX

You may have noticed that the **start()** method has a single parameter called **primaryStage** which is a **Stage** object.

- A **Stage** object is a special kind of **Window** object.
You can think of your JavaFX application as a theatrical performance, where everything happens on the stage.
- In order to make something visible, we must create a **Scene** object



Continue: Basic Structure of JavaFX

A **scene** is an object that can be created using the constructor:

Scene(node, width, height)

This constructor specify the width and height of the scene. (create a window with certain size that will hold the nodes).

Nodes(actors) that perform in the scenes. Panes, groups, controls, and shapes are nodes.

Panes, UI Controls, and Shapes

- A **node** is a visual component such as: a Shape, an Image view, a UI Control, a Group or a Pane.
- A **Shape** refers to a text, line, circle, ellipse, rectangle, etc.
- A **Pane** is a container class that is used for automatically laying out the nodes in a desired location and size.
- A **Group** is a container that groups a collection of nodes. You can apply transformation or effects to a group, which automatically apply to all the children of the group.
- A **UI Control** refers to a label, button, check box, radio button, text field, text area, and so on.

Building GUI

1. Create a class that **extends Application** class.
2. Override the start method that will hold all your Java statements.
 - **public void start(Stage primaryStage){.....}**
3. Create your nodes, UI controls: buttons, labels, etc.
 - **Button btOK = new Button("OK");**
4. Create and Place nodes in a scene.
 - **Scene scene = new Scene(btOK, 200, 250);**
5. Place the Scene in a stage. **primaryStage.setScene(scene);**
6. Set Stage title. **primaryStage.setTitle("MyJavaFX");**
7. Display the stage. **primaryStage.show();**
8. Write **Application.launch(args);** in the main method to run the application.

Needed Java Libraries to build JavaFX GUI

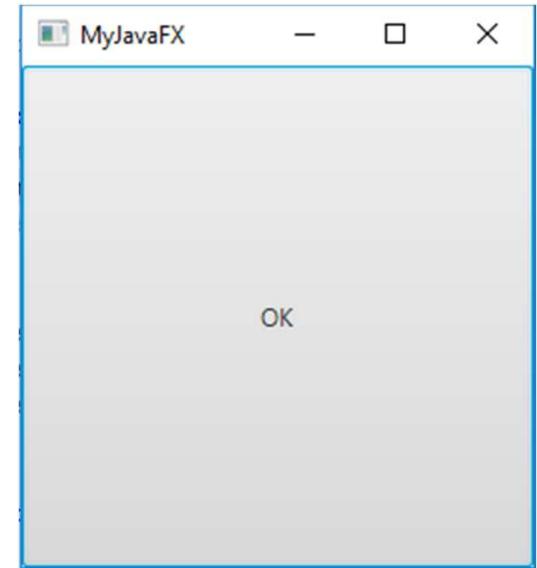
- import javafx.application.Application;
- import javafx.scene.Scene;
- import javafx.scene.control.*;
- import javafx.stage.Stage;

Example(1)

- `public class MyJavaFX extends Application {`
- `@Override // Override the start method in the Application`
- `public void start(Stage primaryStage) {`
- `// Create a button and place it in the scene`
- `Button btOK = new Button("OK");`
- `Scene scene = new Scene(btOK, 200, 250);`
- `primaryStage.setTitle("MyJavaFX"); // Set the stage title`
- `primaryStage.setScene(scene); // Place the scene in the stage`
- `primaryStage.show(); // Display the stage`
- `}`
- `/** * The main method is only needed for the IDE with limited * JavaFX support. Not needed
for running from the command line. */`
- `public static void main(String[] args) { launch(args); }`
- `}`

OutPut

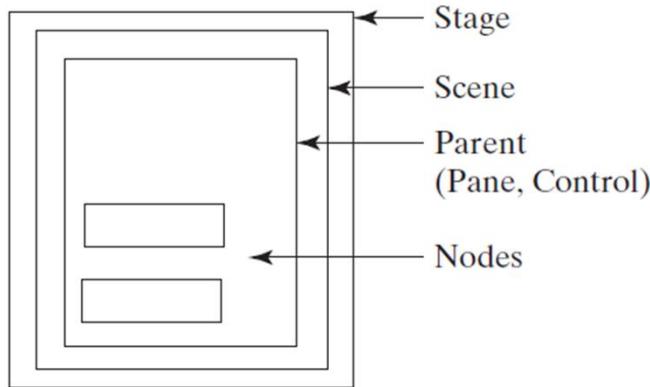
- Note that: The displayed button is always centered in the scene and occupies the entire window.
- You can use container classes called **Panes**, for automatically laying out the nodes in a desired location and size.
- You need to:
 1. place nodes inside a pane then
 2. place the Pane into the Scene.



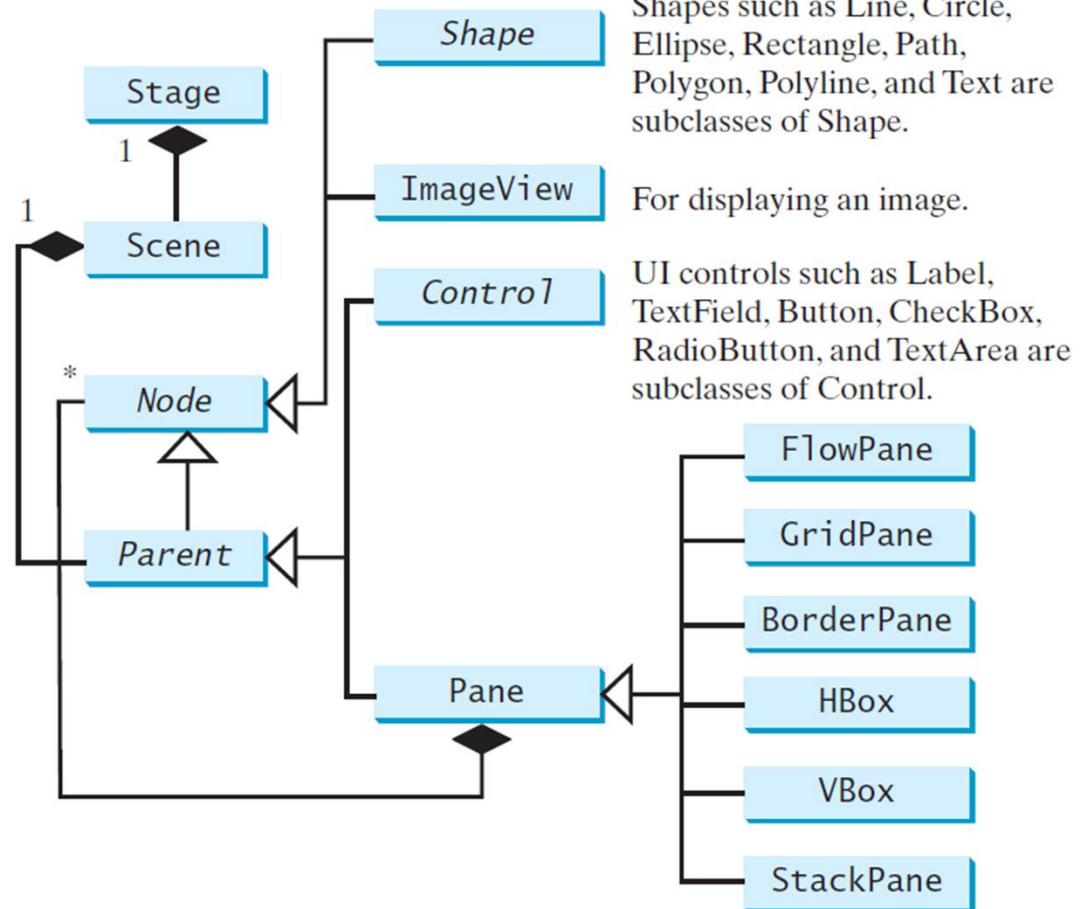
Panes, UI Controls, and Shapes

A Scene can contain a control, Group, or a Pane, but not a Shape or an ImageView.

A Pane or a Group can contain any subtype of Node.



(a)



(b)

Shapes such as Line, Circle, Ellipse, Rectangle, Path, Polygon, Polyline, and Text are subclasses of Shape.

For displaying an image.

UI controls such as Label, TextField, Button, CheckBox, RadioButton, and TextArea are subclasses of Control.

FlowPane
GridPane
BorderPane
HBox
VBox
StackPane

Binding Properties

JavaFX introduces a new concept called *binding property* that enables a *target object* to be bound to a *source object*. If the value in the source object changes, the target property is also changed automatically. The target object is simply called a *binding object* or a *binding property*.

Ex: `label1.textProperty().bind(text1.textProperty());`

With this binding in place, the text displayed by `label1` is automatically updated, character by character, when the user types data into the text field.

JavaFX provides many types of panes for organizing nodes in a container.

<i>Class</i>	<i>Description</i>
Pane	Base class for layout panes. It contains the getChildren() method for returning a list of nodes in the pane.
StackPane	Places the nodes on top of each other in the center of the pane.
FlowPane	Places the nodes row-by-row horizontally or column-by-column vertically.
GridPane	Places the nodes in the cells in a two-dimensional grid.
BorderPane	Places the nodes in the top, right, bottom, left, and center regions.
HBox	Places the nodes in a single row.
VBox	Places the nodes in a single column.

FlowPane

FlowPane: arrange the nodes in the pane horizontally from left to right (Default), or vertically from top to bottom, in the order in which they were added.

javafx.scene.layout.FlowPane

```
-alignment: ObjectProperty<Pos>
-orientation:
    ObjectProperty<Orientation>
-hgap: DoubleProperty
-vgap: DoubleProperty

+FlowPane()
+FlowPane(hgap: double, vgap:
    double)
+FlowPane(orientation:
    ObjectProperty<Orientation>)
+FlowPane(orientation:
    ObjectProperty<Orientation>,
    hgap: double, vgap: double)
```

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The overall alignment of the content in this pane (default: Pos.LEFT).
The orientation in this pane (default: Orientation.HORIZONTAL).

The horizontal gap between the nodes (default: 0).
The vertical gap between the nodes (default: 0).

Creates a default FlowPane.

Creates a FlowPane with a specified horizontal and vertical gap.

Creates a FlowPane with a specified orientation.

Creates a FlowPane with a specified orientation, horizontal gap and vertical gap.

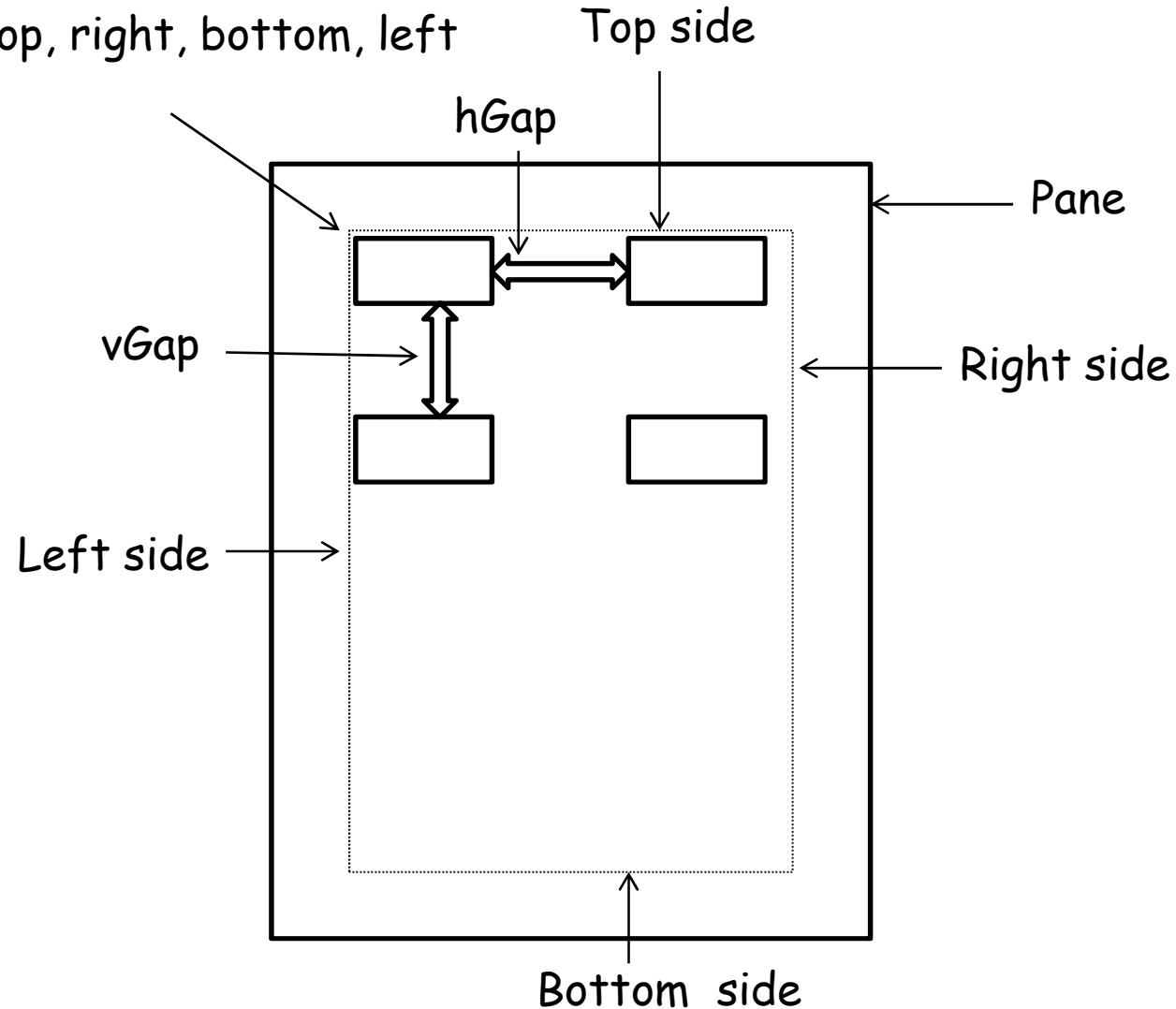
- You can specify the way the nodes placed either by using:
 - `Orientation.HORIZONTAL`: placed horizontally (default)
 - `Orientation.VERTICAL`: the nodes are placed vertically.
- You can specify the gap between nodes in pixels if needed using setter and getter of attributes: `Hgap` or `Vgap`

FlowPane Attributes

- Attributes of FlowPane class: alignment, orientation, hgap, vgap are binding properties.
- Each of which has set and get method. `setHgap(double)`, `getHgap()`, etc.

FlowPane Attributes

Inset: top, right, bottom, left

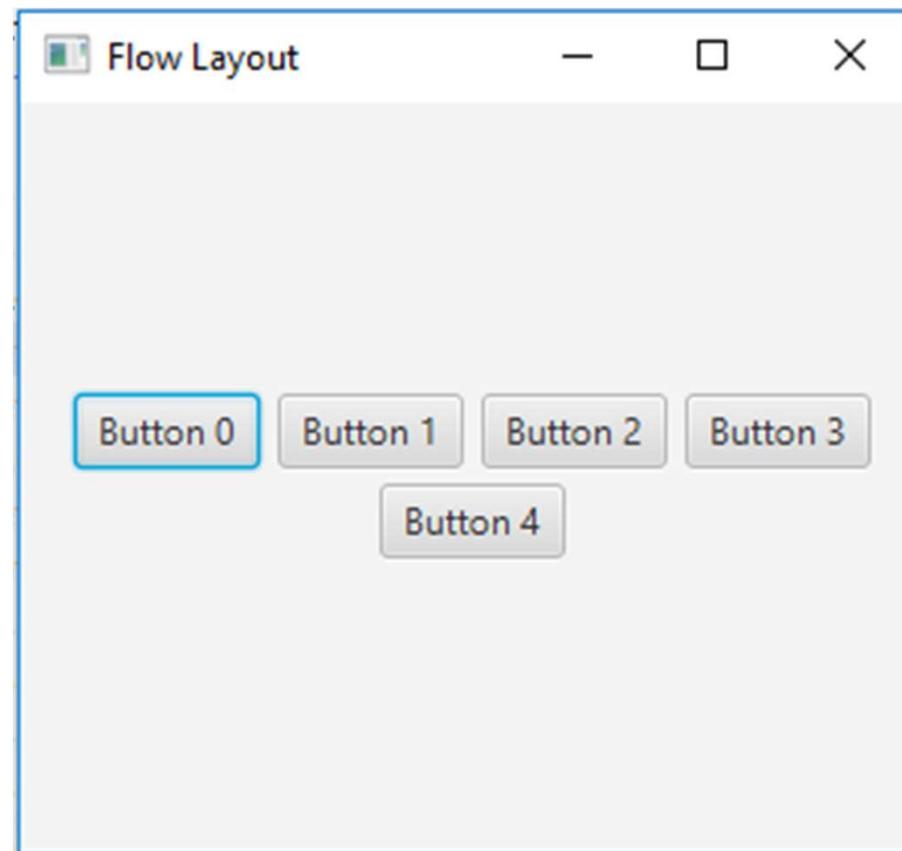


FlowPane_Layout Example

```
public class FlowPane extends Application {  
    public void start(Stage primaryStage) {  
        Button[] b = new Button[5];  
        for(int i=0; i< b.length; i++)  
            b[i] = new Button("Button "+ i);  
        FlowPane root = new FlowPane();  
        root.setPadding(new Insets(11,12,13,14));  
        root.setAlignment(Pos.CENTER);  
        //nodes are placed in center of FlowPane  
        root.setHgap(6);  
        root.setVgap(5);  
        root.getChildren().add(b[0]);  
        root.getChildren().add(b[1]);  
        root.getChildren().add(b[2]);  
        root.getChildren().add(b[3]);  
        root.getChildren().add(b[4]);  
    }  
}
```

```
Scene scene = new Scene(root, 300, 250);  
primaryStage.setTitle("Flow Layout");  
primaryStage.setScene(scene);  
primaryStage.show();  
}  
  
public static void main(String[] args) {  
    launch(args);  
}  
} //End of class
```

FlowPane_Layout Example Output



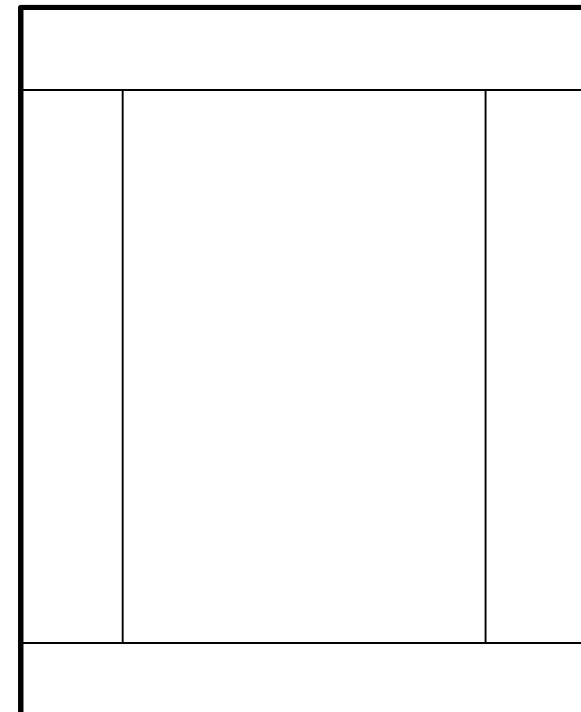
```
root.getChildren().add(b[i]);
```

- The `getchildren()` method returns an instance of JavaFx List behaves similar to ArrayList for storing a collection of elements.
- Invoking `add(b[i])`: adds button into the list.

BorderPane

- BorderPane: can place nodes in five regions: top, bottom, left, right and center, using the setTop(node), setBottom(node), setLeft(node), setRight(node), and setCenter(node).

setAlignment() - This method is used to set the alignment of the nodes belonging to this pane. This method accepts a node and a priority value.



BorderPane

javafx.scene.layout.BorderPane

-top: ObjectProperty<Node>
-right: ObjectProperty<Node>
-bottom: ObjectProperty<Node>
-left: ObjectProperty<Node>
-center: ObjectProperty<Node>

+BorderPane()

+setAlignment(child: Node, pos: Pos)

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The node placed in the top region (default: null).
The node placed in the right region (default: null).
The node placed in the bottom region (default: null).
The node placed in the left region (default: null).
The node placed in the center region (default: null).

Creates a BorderPane.

Sets the alignment of the node in the BorderPane.

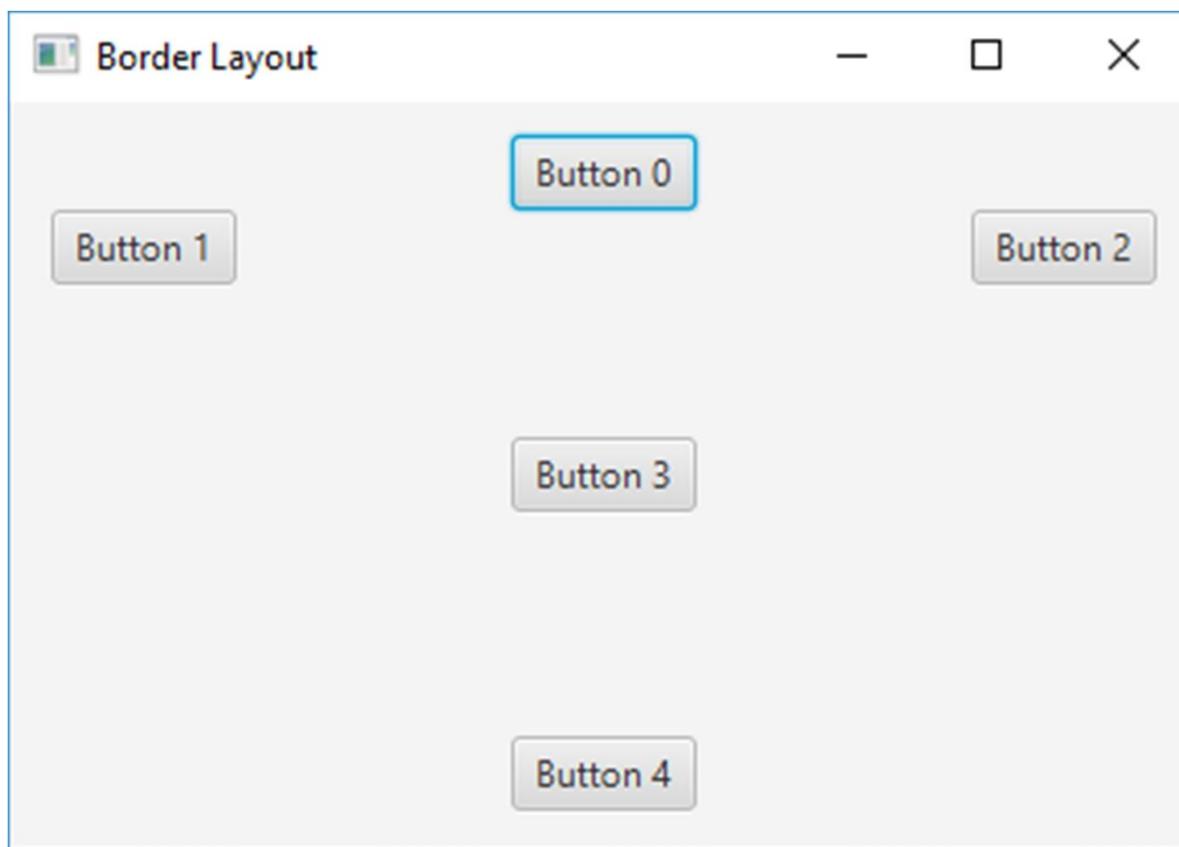
BorderPane Example

```
• public class BorderPane extends Application {  
•     public void start(Stage primaryStage) {  
•         Button[] b = new Button[5];  
•         for(int i=0; i< b.length; i++)  
•             b[i] = new Button("Button "+ i);  
•         BorderPane root = new BorderPane();  
•         root.setPadding(new Insets(11,12,13,14));  
•         root.setAlignment(b[0],Pos.TOP_CENTER);  
•         root.setAlignment(b[4],Pos.BOTTOM_CENTER);
```

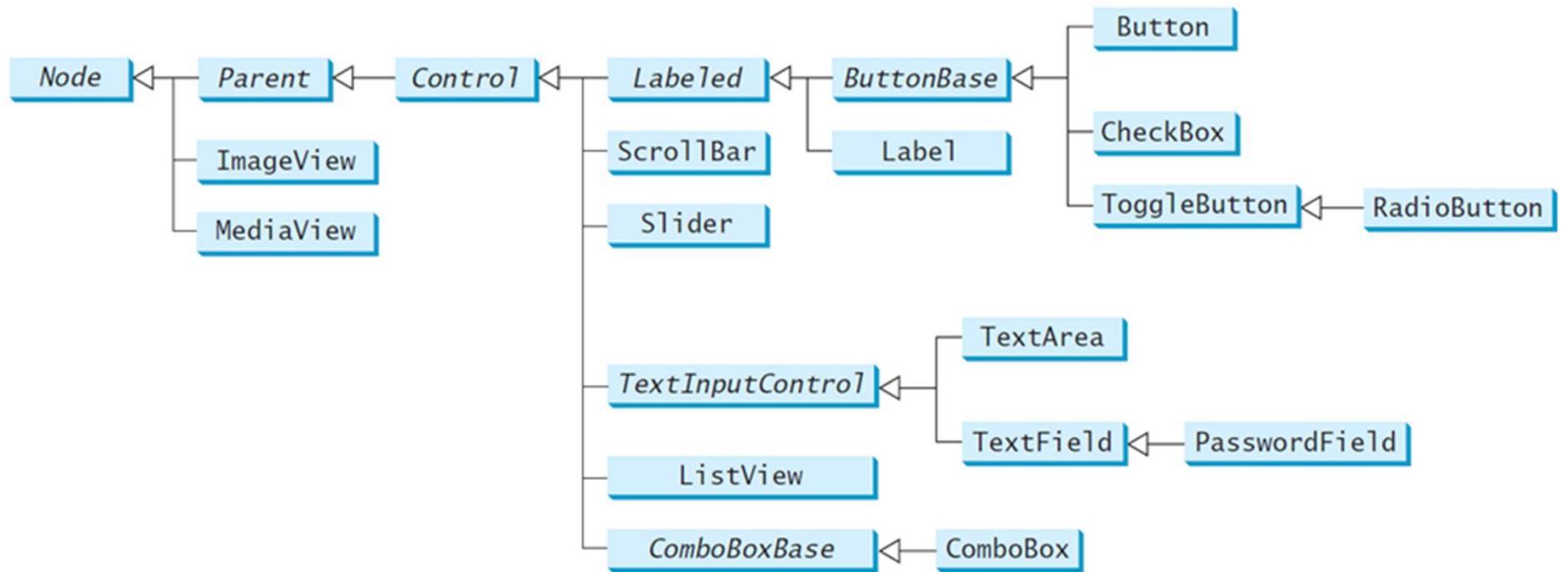
```
root.setTop(b[0]);  
root.setLeft(b[1]);  
root.setRight(b[2]);  
root.setCenter(b[3]);  
root.setBottom(b[4]);
```

```
Scene scene = new Scene(root, 200, 250);  
primaryStage.setTitle("Border Layout");  
primaryStage.setScene(scene);  
primaryStage.show();  
}  
  
public static void main(String[] args) {  
    launch(args);  
}  
}
```

Output Border Layout



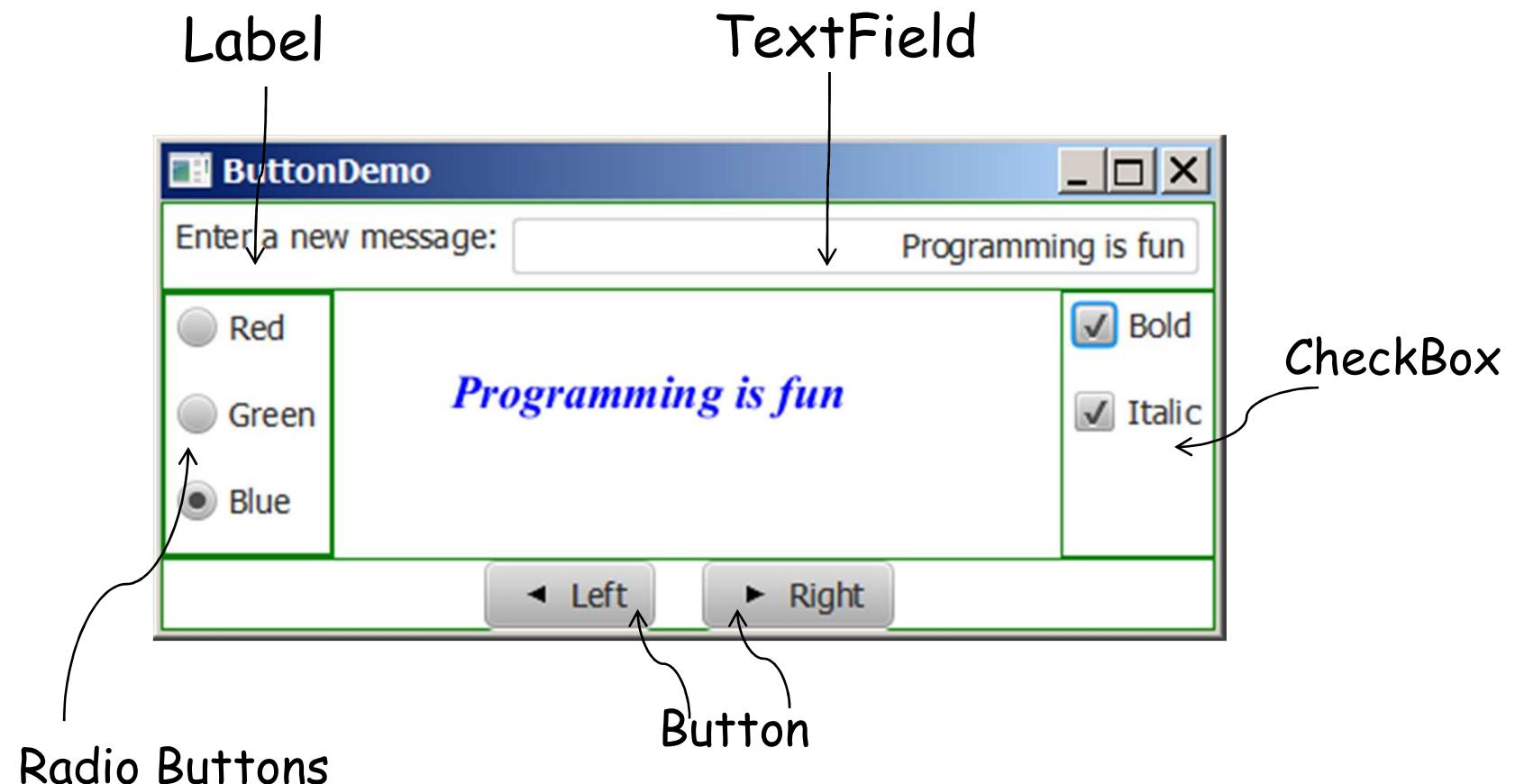
Frequently Used UI Controls



The following UI Controls are used frequently:

`Label`, `Button`, `CheckBox`, `RadioButton`, `TextField`, `PasswordField`,
`TextArea`, `ComboBox`, `ListView`, `ScrollBar` and `MediaPlayer`.

Frequently Used UI Controls



Labeled

A *label* is a display area for a short text, a node, or both. It is often used to label other controls (usually text fields). Labels and buttons share many common properties. These common properties are defined in the **Labeled** class.

- How to create:
- `Label lb = new Label("Hello there");`
- • Setting: To set the text on the label `setText()`
- `lb.setText("Good Bye");`
- • Getting: To get the text from a label `getText()`
- `String currentLabelStr = lb.getText();`

ButtonBase and Button

A *button* is a control that triggers an action event when clicked. JavaFX provides regular buttons, toggle buttons, check box buttons, and radio buttons. The common features of these buttons are defined in **ButtonBase** and **Labeled** classes.

- How to create:
 - `Button offButton = new Button(); // without label`
 - `Button onButton = new Button("Cancel"); //with label`
- Setting: To set the label of a button `setText()`
 - `offButton.setText("Press Me");`
- Getting: To get the label of a button `getText()`
 - `String offButtonLabel = offButton.getText();`

TextField

A text field can be used to enter or display a string. **TextField** is a subclass of **TextInputControl**.

- ❑ How to create by using different overloading constructors:

- `TextField txfA = new TextField();` // empty textbox
- `TextField txfC = new TextField("Type here");` // not empty

- ❑ Setting: To set the label of a button `setText()`

- `txfA.setText("Type here");`

- ❑ Setting: To set the width of text Field

- `txfA.setPrefColumnCount(15);`

- ❑ Getting: To get the label of a button `getText()`

- `String t = txfA.getText();`

TextArea

A **TextArea** enables the user to enter multiple lines of text.

□ How to create:

- **TextArea ta = new TextArea (); // empty**
- **TextArea ta = new TextArea ("This is Text Area");**

□ Determine number of rows and column1s

- **ta.setPrefColumnCount(20);**
- **ta.setPrefRowCount(5);**

□ Setting & Getting

- **same as textbox setText(), getText()**

CheckBox

- They are used for the input of boolean value for a program.
- They can either on or off
- • How to create:
- `CheckBox noviceUserType = new CheckBox("Novice");`
- `CheckBox expeUserType = new CheckBox("Experienced");`
- • Getting: The state of a check box - whether or not it is on - can be discovered by means of the method `isSelected`, which returns a boolean result true or false.
- `boolean b = expeUserType.isSelected();`

RadioButton

Radio buttons, also known as *option buttons*, enable you to choose a single item from a group of choices.

In appearance radio buttons resemble check boxes, but check boxes display a square that is either checked or blank, whereas radio buttons display a circle that is either filled (if selected) or blank (if not selected).

How to create:

// (1) Create the buttons

```
RadioButton fr = new RadioButton("French");
RadioButton en = new RadioButton("English");
sn.setSelected(true); //initially selected
```

// (2) Group the buttons

```
ToggleGroup languageGroup = new ToggleGroup();
fr.setToggleGroup(languageGroup);
en.setToggleGroup(languageGroup);
```

Note: The aim of using a `ToggleGroup` object is to tell Java that a set of buttons are grouped together and have the property that only one of the buttons can be selected at a time, that means the radio button that was selected previously becomes unselected.

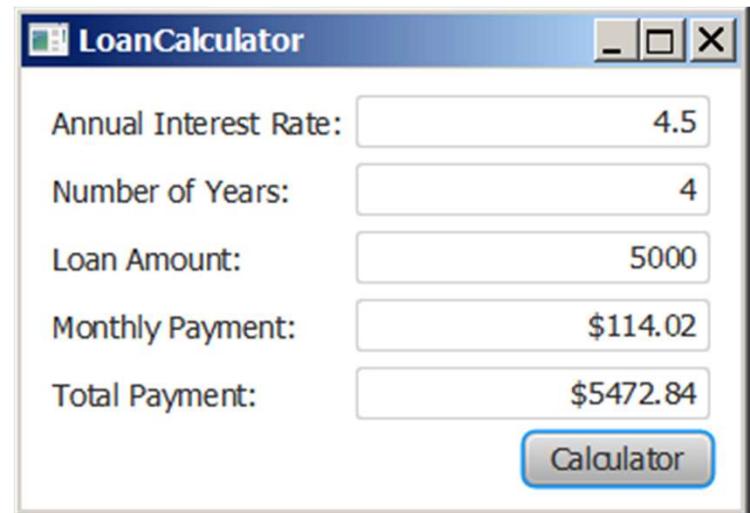
Event Driven Programming

Suppose you want to write a GUI program that:

- lets the user enter a loan amount, annual interest rate, and number of years.
- then click the *Compute Payment* button to obtain the monthly payment and total payment.

How do you accomplish the task?

You have to use event-driven programming to write the code to respond to the button-clicking event.



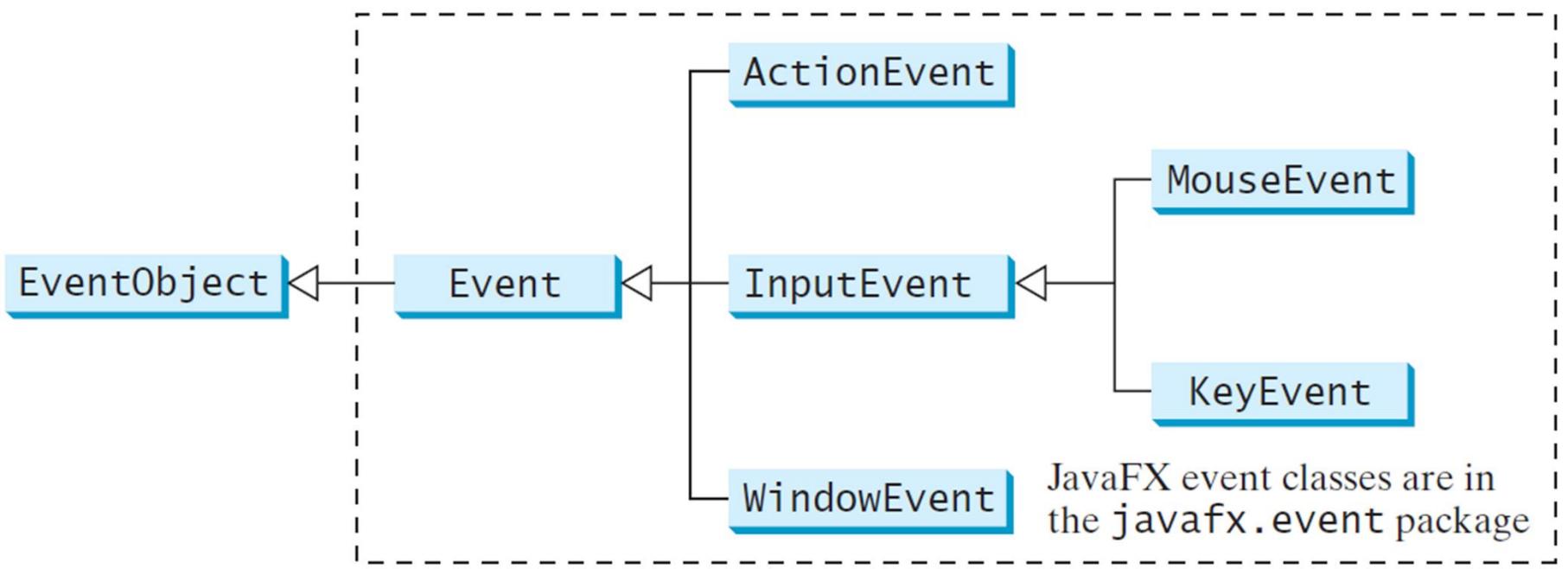
Procedural vs. Event-Driven Programming

- Procedural programming is executed in procedural order.
- In event-driven programming, code is executed upon activation of events.

Events

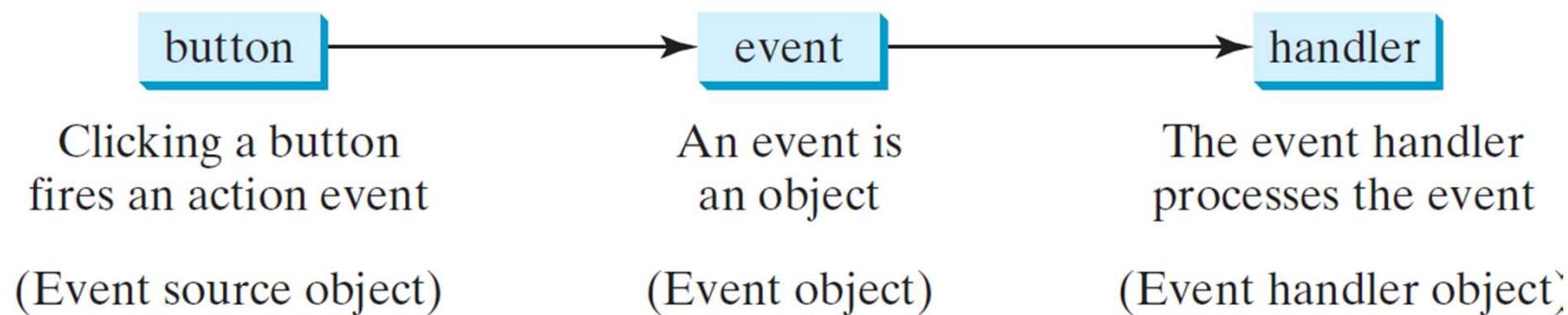
- An **event** can be defined as a type of signal to the program that something has happened.
- The event is generated by external user actions such as a button click, mouse movements, mouse clicks, or keystrokes.

Event classes



Event Handler

- To respond to a button click, you need to write the code to process the button-clicking action (what will be the result if you press this button). Therefore you have two members in this process:
 - Event source object:** where the object is originated that fires an action such as clicking a button.
 - The event handler or event listener object:** which is an object that you create to handle the action event on a button, it contains a method for processing the event.



Registering Handlers & Handling events

- An event handler object should do two things to handle an event:
 1. You need to create a handler object as an instance of the corresponding event-handler interface to ensure the handler has the correct method for processing the event. JavaFx defines a unified handler interface `EventHandler<T extends Event>` for an event T, where T is a generic type that is a sub type of event. The handler interface contains the `handle(T e)` method for handling events.
 2. The handler object must be registered by the source object. Registration methods depend on the event type.
 - For an action event, the method `setOnAction()`.
 - For a mouse-pressed event, the method is `setOnMousePressed()`.
 - For a keypressed event, the method is `setOnKeyPressed()`.

Selected User Actions and Handlers

User Action	Source Object	Event Type Fired	Event Registration Method
Click a button	<code>Button</code>	<code>ActionEvent</code>	<code>setOnAction(EventHandler<ActionEvent>)</code>
Press Enter in a text field	<code>TextField</code>	<code>ActionEvent</code>	<code>setOnAction(EventHandler<ActionEvent>)</code>
Check or uncheck	<code>RadioButton</code>	<code>ActionEvent</code>	<code>setOnAction(EventHandler<ActionEvent>)</code>
Check or uncheck	<code>CheckBox</code>	<code>ActionEvent</code>	<code>setOnAction(EventHandler<ActionEvent>)</code>
Select a new item	<code>ComboBox</code>	<code>ActionEvent</code>	<code>setOnAction(EventHandler<ActionEvent>)</code>
Mouse pressed	<code>Node, Scene</code>	<code>MouseEvent</code>	<code>setOnMousePressed(EventHandler<MouseEvent>)</code>
Mouse released			<code>setOnMouseReleased(EventHandler<MouseEvent>)</code>
Mouse clicked			<code>setOnMouseClicked(EventHandler<MouseEvent>)</code>
Mouse entered			<code>setOnMouseEntered(EventHandler<MouseEvent>)</code>
Mouse exited			<code>setOnMouseExited(EventHandler<MouseEvent>)</code>
Mouse moved			<code>setOnMouseMoved(EventHandler<MouseEvent>)</code>
Mouse dragged			<code>setOnMouseDragged(EventHandler<MouseEvent>)</code>
Key pressed	<code>Node, Scene</code>	<code>KeyEvent</code>	<code>setOnKeyPressed(EventHandler<KeyEvent>)</code>
Key released			<code>setOnKeyReleased(EventHandler<KeyEvent>)</code>
Key typed			<code>setOnKeyTyped(EventHandler<KeyEvent>)</code>

Handling events

- There are three ways to handle an event using:
 1. Inner classes handler
 2. Anonymous Inner-Class handlers
 3. Using Lambda Expression

1- Inner Classes

- **An inner class**, or nested class, is a class defined within a scope of another class.
Inner classes are useful for defining handler classes.
- **Advantages:** In some applications, you can use an inner class to make programs simple.

1- Inner Classes-Cont

```
public class Test {  
    ...  
}  
  
public class A {  
    ...  
}
```

(a)

```
public class Test {  
    ...  
  
    // Inner class  
    public class A {  
        ...  
    }  
}
```

(b)

```
// OuterClass.java: inner class demo  
public class OuterClass {  
    private int data;  
  
    /** A method in the outer class */  
    public void m() {  
        // Do something  
    }  
  
    // An inner class  
    class InnerClass {  
        /** A method in the inner class */  
        public void mi() {  
            // Directly reference data and method  
            // defined in its outer class  
            data++;  
            m();  
        }  
    }  
}
```

(c)

Using Inner class to handle an event

```
public class HandleEvent extends Application {  
    public void start(Stage primaryStage) {  
        ...  
        OKHandlerClass handler1 = new OKHandlerClass();  
        btOK.setOnAction(handler1);  
        CancelHandlerClass handler2 = new CancelHandlerClass();  
        btCancel.setOnAction(handler2);  
        ...  
        primaryStage.show(); // Display the stage  
    }  
}
```

1. Start from the main method to create a window and display it

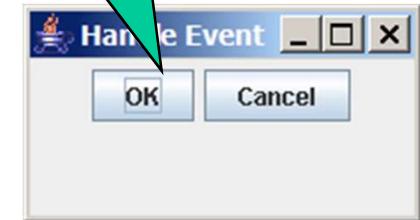


```
class OKHandlerClass implements EventHandler<ActionEvent> {  
    @Override  
    public void handle(ActionEvent e) {  
        System.out.println("OK button clicked");  
    }  
}
```

Trace Execution

```
public class HandleEvent extends Application {  
    public void start(Stage primaryStage) {  
        ...  
        OKHandlerClass handler1 = new OKHandlerClass(); // create a handler object  
        btOK.setOnAction(handler1); // register handler object with the source object  
  
        CancelHandlerClass handler2 = new CancelHandlerClass();  
        btCancel.setOnAction(handler2);  
        ...  
        primaryStage.show(); // Display the stage  
    }  
}  
  
class OKHandlerClass implements EventHandler<ActionEvent> { //inner class  
    @Override  
    public void handle(ActionEvent e) {  
        System.out.println("OK button clicked");  
    }  
}
```

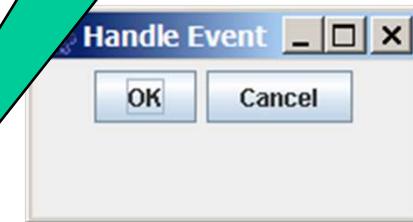
2. Click OK



Trace Execution

```
public class HandleEvent extends Application {  
    public void start(Stage primaryStage) {  
        ...  
        OKHandlerClass handler1 = new OKHandlerClass();  
        btOK.setOnAction(handler1);  
        CancelHandlerClass handler2 = new CancelHandlerClass();  
        btCancel.setOnAction(handler2);  
        ...  
        primaryStage.show(); // Display the stage  
    }  
}  
  
class OKHandlerClass implements EventHandler<ActionEvent> {  
    @Override  
    public void handle(ActionEvent e) {  
        System.out.println("OK button clicked");  
    }  
}
```

3. The JVM invokes the listener's handle method



getSource() method

Instead of creating two different inner classes for two buttons, we can identify the source of event using `getSource()` method.

Using getSource()

```
public class HandleEvent extends Application {  
    public void start(Stage primaryStage) {  
        ...  
        HandlerClass handler1 = new HandlerClass();  
        btOK.setOnAction(handler1);  
        HandlerClass handler2 = new HandlerClass();  
        btCancel.setOnAction(handler2);  
        ...  
        primaryStage.show(); // Display the stage  
    }  
}  
class HandlerClass implements EventHandler<ActionEvent> {  
    @Override  
    public void handle(ActionEvent e) {  
        if(e.getSource().equals(btOK))  
            System.out.println("OK button clicked");  
        else if( e.getSource().equals(btCancel))  
            .....  
    }  
}
```

Using Anonymous Inner Class to handle an event

- An anonymous inner class is an inner class without a name. It combines defining an inner class and creating an instance of the class into one step.
- The syntax for anonymous inner class is:

```
new SuperClassName/InterfaceName() {  
    // Implement or override methods in superclass or interface  
    // Other methods if necessary  
}
```

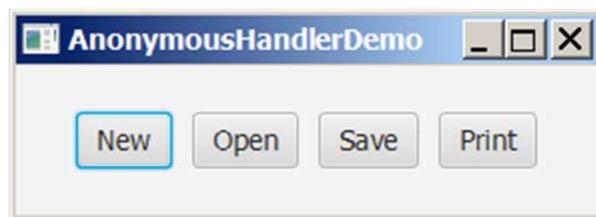
Using Anonymous Inner Class to handle an event

```
public void start(Stage primaryStage) {  
    // Omitted  
  
    btEnlarge.setOnAction(  
        new EnlargeHandler());  
}  
  
class EnlargeHandler  
    implements EventHandler<ActionEvent> {  
    public void handle(ActionEvent e) {  
        circlePane.enlarge();  
    }  
}
```

(a) Inner class EnlargeListener

```
public void start(Stage primaryStage) {  
    // Omitted  
  
    btEnlarge.setOnAction(  
        new class EnlargeHandler  
            implements EventHandler<ActionEvent>() {  
                public void handle(ActionEvent e) {  
                    circlePane.enlarge();  
                }  
            });  
}
```

(b) Anonymous inner class



Using Anonymous Inner Class to handle an event

```
//From previous example
```

```
btOK.setOnAction( new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent e){  
        System.out.println("OK button clicked");  
    } // method  
} // event Handler  
); //argument
```

```
// You did registration and creation on one step
```

Simplifying Event Handling Using Lambda Expressions

Lambda expression is a new feature in Java 8. Lambda expressions can be viewed as an anonymous method with a concise syntax.

For example, the following code in (a) can be greatly simplified using a lambda expression in (b) in three lines.

```
btEnlarge.setOnAction(  
    new EventHandler<ActionEvent>() {  
        @Override  
        public void handle(ActionEvent e) {  
            // Code for processing event e  
        }  
    }  
) ;
```

(a) Anonymous inner class event handler

```
btEnlarge.setOnAction(e -> {  
    // Code for processing event e  
});
```

(b) Lambda expression event handler

Lambda Expressions

//From previous example

```
btOK.setOnAction(  
    e->{ System.out.println("OK button clicked"); }  
);
```

The MouseEvent Class

`javafx.scene.input.MouseEvent`

```
+getButton(): MouseButton  
+getClickCount(): int  
+getX(): double  
+getY(): double  
+getSceneX(): double  
+getSceneY(): double  
+getScreenX(): double  
+getScreenY(): double  
+isAltDown(): boolean  
+isControlDown(): boolean  
+isMetaDown(): boolean  
+isShiftDown(): boolean
```

Indicates which mouse button has been clicked.

Returns the number of mouse clicks associated with this event.

Returns the x-coordinate of the mouse point in the event source node.

Returns the y-coordinate of the mouse point in the event source node.

Returns the x-coordinate of the mouse point in the scene.

Returns the y-coordinate of the mouse point in the scene.

Returns the x-coordinate of the mouse point in the screen.

Returns the y-coordinate of the mouse point in the screen.

Returns true if the `Alt` key is pressed on this event.

Returns true if the `Control` key is pressed on this event.

Returns true if the mouse `Meta` button is pressed on this event.

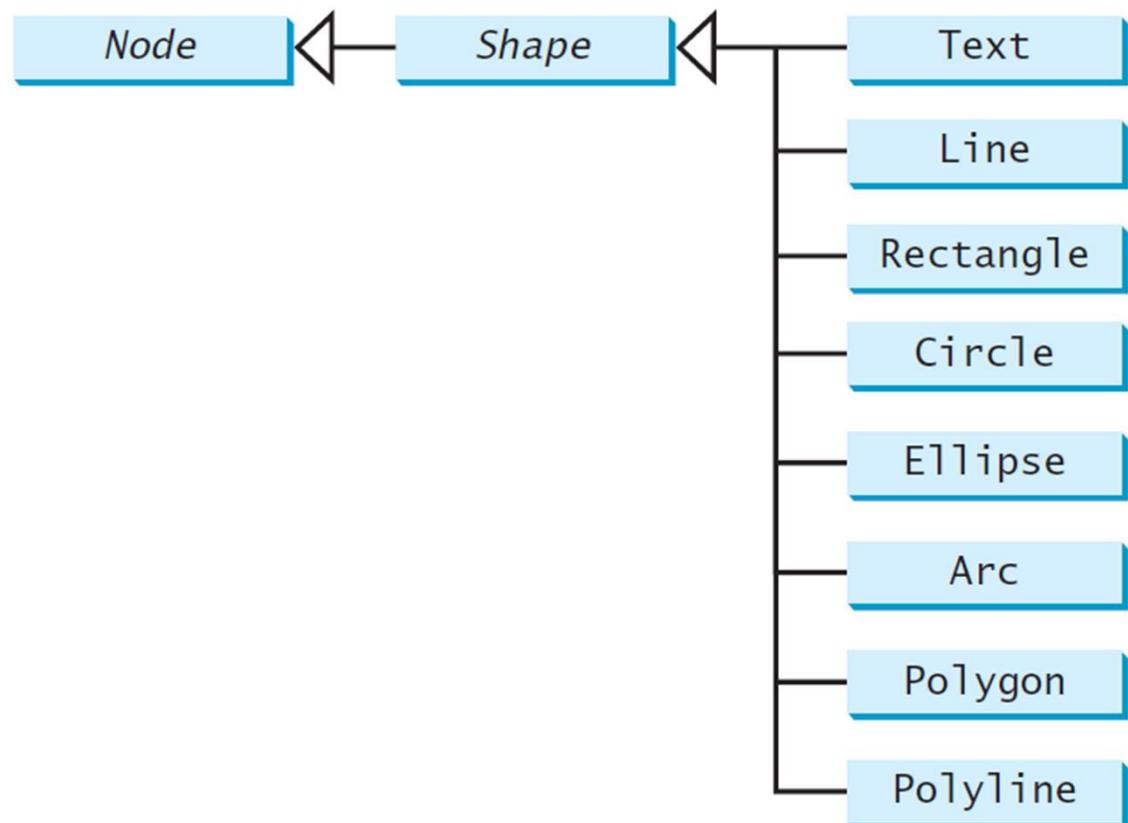
Returns true if the `Shift` key is pressed on this event.

Event Registered Methods

Mouse pressed	Node, Scene	MouseEvent	<code>setOnMousePressed(EventHandler<MouseEvent>)</code>
Mouse released			<code>setOnMouseReleased(EventHandler<MouseEvent>)</code>
Mouse clicked			<code>setOnMouseClicked(EventHandler<MouseEvent>)</code>
Mouse entered			<code>setOnMouseEntered(EventHandler<MouseEvent>)</code>
Mouse exited			<code>setOnMouseExited(EventHandler<MouseEvent>)</code>
Mouse moved			<code>setOnMouseMoved(EventHandler<MouseEvent>)</code>
Mouse dragged			<code>setOnMouseDragged(EventHandler<MouseEvent>)</code>

Shapes

JavaFX provides many shape classes for drawing texts, lines, circles, rectangles, ellipses, arcs, polygons, and polylines.



Text

javafx.scene.text.Text

-text: StringProperty
-x: DoubleProperty
-y: DoubleProperty
-underline: BooleanProperty
-strikethrough: BooleanProperty
-font: ObjectProperty

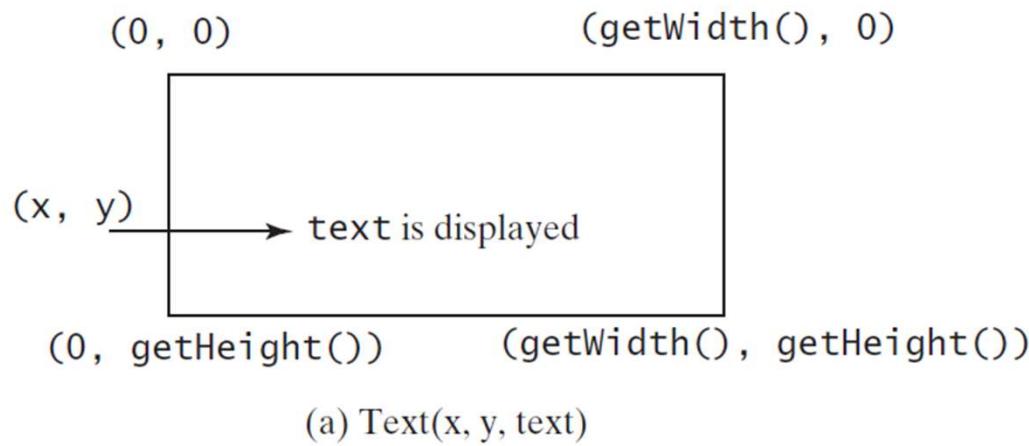
+Text()
+Text(text: String)
+Text(x: double, y: double,
text: String)

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

Defines the text to be displayed.
Defines the x-coordinate of text (default 0).
Defines the y-coordinate of text (default 0).
Defines if each line has an underline below it (default `false`).
Defines if each line has a line through it (default `false`).
Defines the font for the text.

Creates an empty Text.
Creates a Text with the specified text.
Creates a Text with the specified x-, y-coordinates and text.

Text



```
Pane pane = new Pane();
pane.setPadding(new Insets(5, 5, 5, 5));
Text text1 = new Text(20, 20, "Programming is fun");
text1.setFont(Font.font("Courier", FontWeight.BOLD, FontPosture.ITALIC, 15));
pane.getChildren().add(text1);
Text text2 = new Text(60, 60, "Programming is fun\nDisplay text");
pane.getChildren().add(text2);
Text text3 = new Text(10, 100, "Programming is fun\nDisplay text");
text3.setFill(Color.RED); text3.setUnderline(true); text3.setStrikethrough(true);
pane.getChildren().add(text3);
```

Exampe Dragging a Text

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class Mouse extends Application {
    public void start(Stage primaryStage) {
        Pane pane = new Pane();
        Text text = new Text(20, 20, "Programming is fun");
        pane.getChildren().addAll(text);

        text.setOnMouseDragged(e -> {
            text.setX(e.getX());
            text.setY(e.getY());
        });
    }
}
```

```
Scene scene = new Scene(pane, 300, 100);
primaryStage.setTitle("MouseEventDemo");
primaryStage.setScene(scene);
primaryStage.show();
}

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

}
```

Line

javafx.scene.shape.Line

```
-startX: DoubleProperty  
-startY: DoubleProperty  
-endX: DoubleProperty  
-endY: DoubleProperty  
  
+Line()  
+Line(startX: double, startY:  
      double, endX: double, endY:  
      double)
```

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The x-coordinate of the start point.

The y-coordinate of the start point.

The x-coordinate of the end point.

The y-coordinate of the end point.

Creates an empty Line.

Creates a Line with the specified starting and ending points.

(0, 0)

(getWidth(), 0)

(startX, startY)

(endX, endY)

ShowLine

(0, getHeight())

(getWidth(), getHeight())

Rectangle

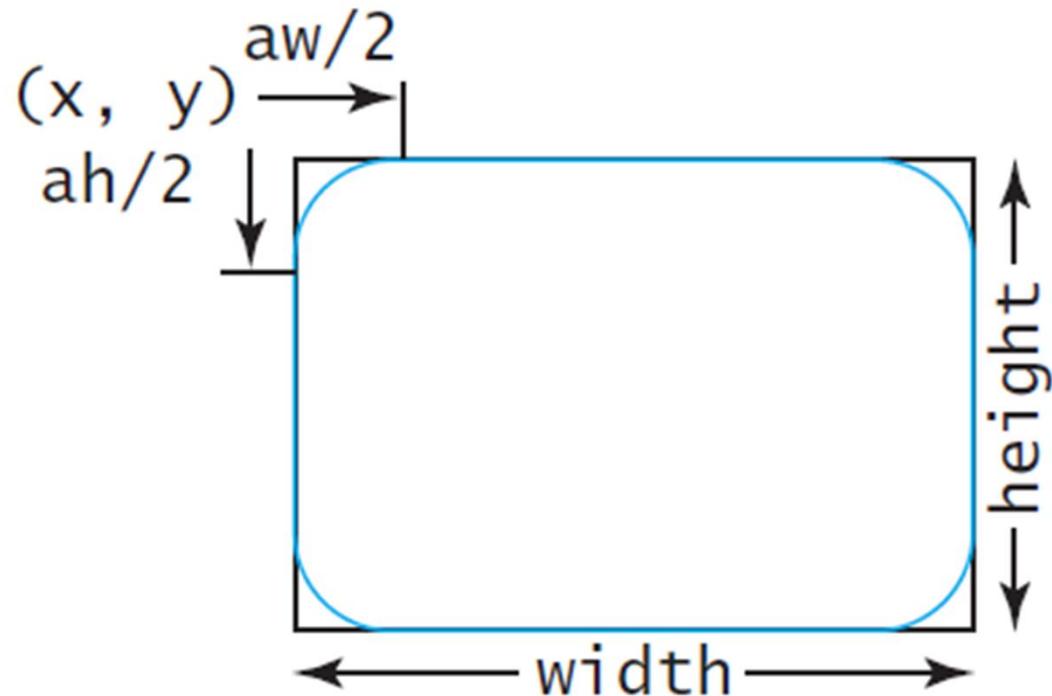
javafx.scene.shape.Rectangle

```
-x: DoubleProperty  
-y: DoubleProperty  
-width: DoubleProperty  
-height: DoubleProperty  
-arcWidth: DoubleProperty  
  
-arcHeight: DoubleProperty  
  
+Rectangle()  
+Rectanlge(x: double, y:  
    double, width: double,  
    height: double)
```

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The x-coordinate of the upper-left corner of the rectangle (default 0).
The y-coordinate of the upper-left corner of the rectangle (default 0).
The width of the rectangle (default: 0).
The height of the rectangle (default: 0).
The **arcWidth** of the rectangle (default: 0). **arcWidth** is the horizontal diameter of the arcs at the corner (see Figure 14.31a).
The **arcHeight** of the rectangle (default: 0). **arcHeight** is the vertical diameter of the arcs at the corner (see Figure 14.31a).
Creates an empty **Rectangle**.
Creates a **Rectangle** with the specified upper-left corner point, width, and height.

Rectangle Example



(a) `Rectangle(x, y, w, h)`

ShowRectangle

Circle

javafx.scene.shape.Circle

```
-centerX: DoubleProperty  
-centerY: DoubleProperty  
-radius: DoubleProperty  
  
+Circle()  
+Circle(x: double, y: double)  
+Circle(x: double, y: double,  
       radius: double)
```

The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

The x-coordinate of the center of the circle (default 0).
The y-coordinate of the center of the circle (default 0).
The radius of the circle (default: 0).

Creates an empty **Circle**.

Creates a **Circle** with the specified center.

Creates a **Circle** with the specified center and radius.

End of the chapter ...