Introduction

- JavaFX is a new framework for developing Java GUI programs.
- Today, we introduce:
  - The framework of JavaFX
  - The JavaFX GUI components and their relationships
- You will learn how to develop simple GUI programs using buttons, labels, text fields, colors, fonts.

JavaFX vs Swing and AWT

- Swing and AWT are replaced by the JavaFX platform for developing rich Internet applications.
- AWT is fine for developing simple graphical user interfaces, but not for developing comprehensive GUI projects.
- The AWT user-interface components were replaced by a more robust, versatile, and flexible library known as Swing components.
- Swing is now replaced by a completely new GUI platform known as JavaFX.

JavaFX vs Swing and AWT (Cont.)

- JavaFX incorporates modern GUI technologies to enable us to develop rich Internet applications (RIA).
- RIA is a Web application designed to deliver the same features and functions normally associated with desktop applications.
- While swing is designed for developing desktop GUI applications, JavaFX applications can run on a desktop and from a Web browser.
- Additionally, JavaFX provides a multi-touch support for touch-enabled devices such as tablets and smart phones.
- JavaFX has a built-in 2D, 3D, animation support, video and audio playback, and runs as a stand-alone application or from a browser.

The Basic Structure of a JavaFX Program

- The abstract javafx.application.Application class defines the essential framework for writing JavaFX programs.
- Let’s begin by writing a simple JavaFX program that illustrates the basic structure of a JavaFX program.
- As you will see in our first example, Every JavaFX program is defined in a class that extends **javafx.application.Application**

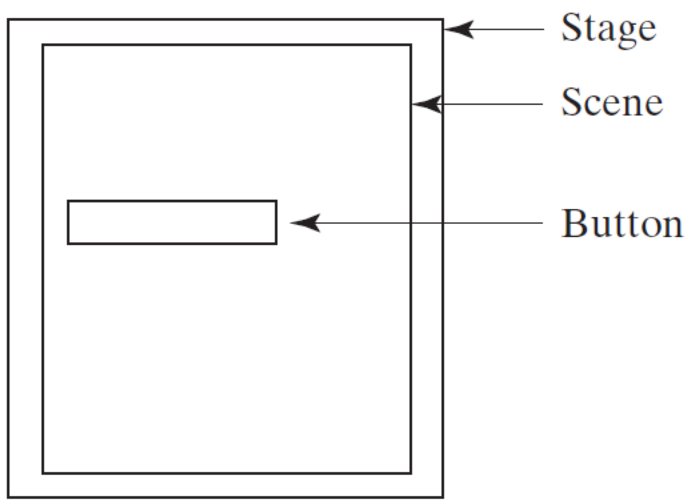
`MyJavaFX.java`

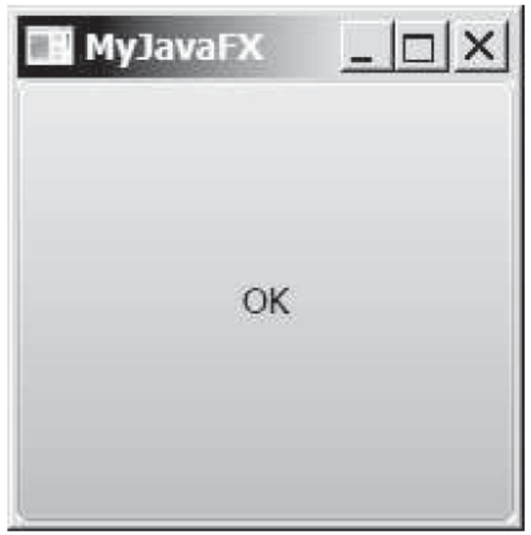
Example - Explanations

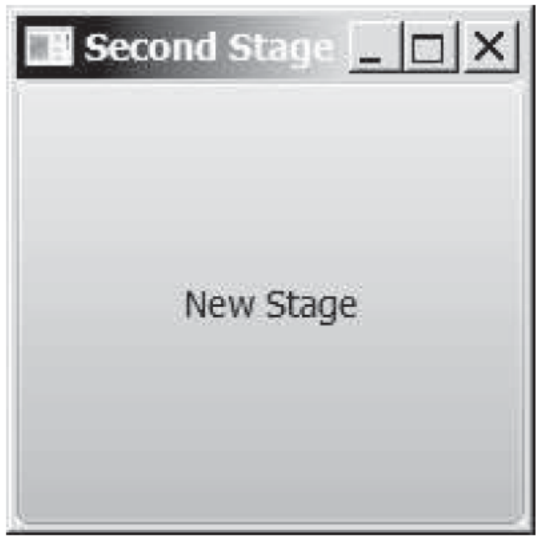
- The launch method is a static method defined in the Application class for launching a stand-alone JavaFX application.
- The main class overrides the start method defined in javafx.application.Application
- After a JavaFX application is launched, the JVM constructs an instance of the class using its no-arg constructor and invokes its start method.
- The start method normally places UI controls in a scene and displays the scene in a stage, as shown in in the next slide.

Example - Explanations (Cont.)

- Stage is a window for displaying a scene that contains nodes.
- Multiple stages can be displayed in a JavaFX program.







Scene-Object

- The program creates a Button object and places it in a Scene object.
- A Scene object can be created using the constructor Scene(node, width, height).
- This constructor specifies the width and height of the scene and places the node in the scene.

Stage-Object

- A Stage object called primary stage is a window which is automatically created by the JVM when the application is launched.
- The program sets the scene to the primary stage by calling setScene(scene) on the primary stage object
- The program also displays the primary stage by invoking show() on the primary stage object.
- JavaFX names the Stage and Scene classes using the analogy from the theater. You may think stage as the platform to support scenes and nodes as actors to perform in the scenes.

Example 02

- Let’s write a JavaFX program displaying two stages.

`MultipleStageDemo.java`

Example 02 – Explanation

- When you run a JavaFX application without a main method, JVM automatically invokes the launch method to run the application.
- By default, the user can resize the stage. To prevent the user from resizing the stage, invoke stage.setResizable(false).

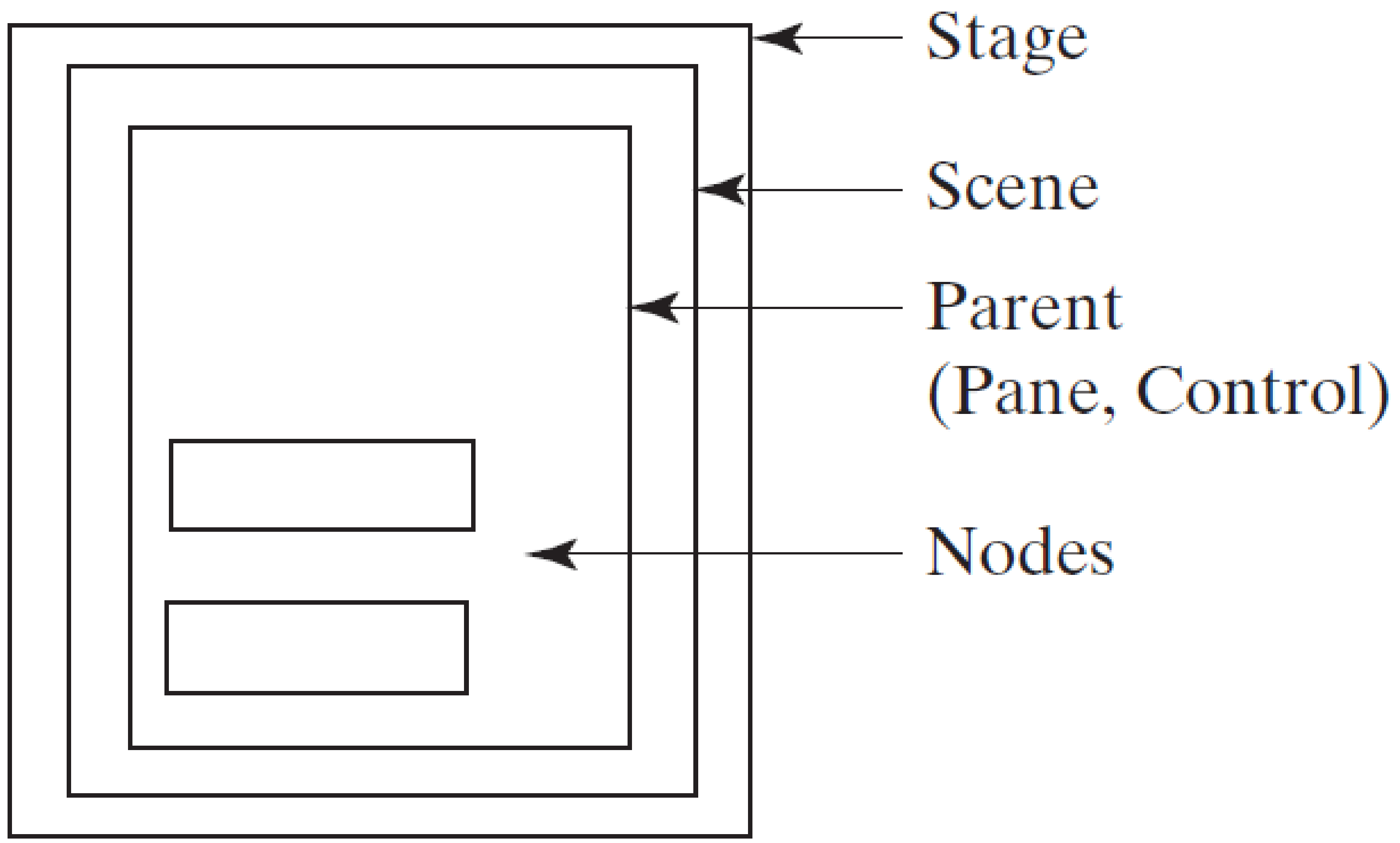
Panes, UI Controls, and Shapes

- *Panes, UI controls, and shapes are subtypes of Node.*
- The button is always centered in the scene and occupies the entire window no matter how you resize it.
- You can fix the problem by setting the position and size properties of a button.
- However, a better approach is to use container classes, called panes, for automatically laying out the nodes in a desired location and size.

Panes, UI Controls, and Shapes (Cont.)

- ➢ You place nodes inside a pane and then place the pane into a scene.
- ➢ A node is a visual component such as a shape, an image view, a UI control, or a pane.
- ➢ A shape refers to a text, line, circle, ellipse, rectangle, arc, polygon, polyline, etc.
- ➢ A UI control refers to a label, button, check box, radio button, text field, text area, etc.
- ➢ A scene can be displayed in a stage, as shown in the next slide.

Panes are Used to Hold Nodes

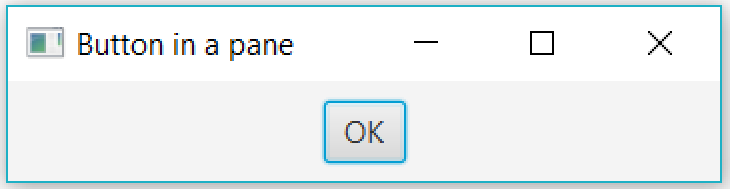


Panes, UI Controls, and Shapes (Cont.)

- Note that a Scene can contain a Control or a Pane, but not a Shape or an ImageView.
- A Pane can contain any subtype of Node.
- You can create a Scene using the constructor Scene(Parent, width, height) or Scene(Parent).

Example 03

- Let's write a program that places a button in a pane as shown below



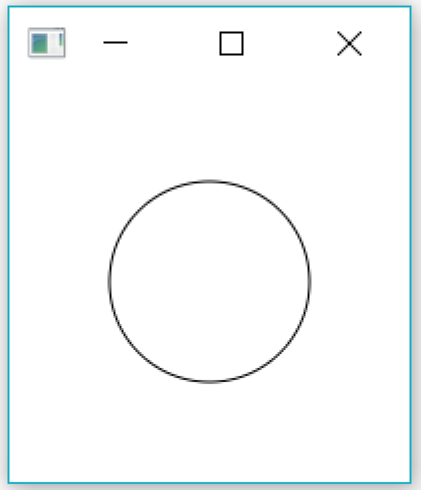
ButtonInPane.java

Example 03 - Explanation

- The program creates a StackPane object by using non-arg constructor and then adds a button as a child of the pane by suing getChildren() and add() methods.
- The getChildren() method returns an instance of javafx.collections.ObservableList.ObservableList behaves very much like an ArrayList for storing a collection of elements.
- Invoking add(e) adds an element to the list.
- The StackPane places the nodes in the center of the pane on top of each other.
- Here, there is only one node in the pane. The StackPane respects a node’s preferred size. So you see the button displayed in its preferred size.

Example 04

- Let’s write an example that displays a circle in the center of the pane, as shown in below



`ShowCircle.java`

Example 04- Explanation

- The circle is displayed in the center of the stage, as shown in the previous slide.
- However, if you resize the window, the circle is not centered.
- In order to display the circle centered as the window resizes, the *x*- and *y*-coordinates of the circle center need to be reset to the center of the pane.
- This can be done by using property binding.

Property Binding

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

Property Binding(Cont.)

- As discussed in the previous example, the circle is not centered after the window is resized
- In order to display the circle centered as the window resizes, the *x*- and *y*-coordinates of the circle center need to be reset to the center of the pane.
- This can be done by binding the **centerX** with pane's **width/2** and **centerY** with pane's **height/2**

Example 05

- Let’s See an example of Property Binding

```
ShowCircleCentered.java
```

Explanation - ShowCircleCentered

- This programs binds **circle**’s **centerX** and **centerY** properties to half of **pane**’s width and height.
- **divide** methods for dividing a value in a binding property and returning a new observable property
- So **pane.widthProperty().divide(2)** returns a new observable property that represents half of the **pane**’s width.

Explanation - ShowCircleCentered

- In this statement
  - circle.centerXProperty().bind(pane.widthProperty().divide(**2**));

- Since **centerX** is bound to **width.divide(2)**, when **pane**’s width is changed, **centerX** automatically updates itself to match **pane**’s width / 2.

Example 06

- Let’s look at another example

`BindingDemo.java`

Example 06 – Explanation

- The program creates an instance of **DoubleProperty** using **SimpleDoubleProperty(1)**

- Note that **DoubleProperty, FloatProperty**, **LongProperty**, **IntegerProperty**, and **BooleanProperty** are abstract classes

- Their concrete subclasses such as **SimpleDoubleProperty**, used to create instances of these properties

Common Properties and Methods for Nodes

- *The* **Node** *defines many properties and methods that are common to all nodes.*

- Let’s look at the two common property for node:
- **style**
- **rotate**

Common Properties and Methods for Nodes

- JavaFX style properties are similar to cascading style sheets (CSS) used to specify the styles for HTML elements in a Web page

- Therefore, the style properties in JavaFX are called *JavaFX CSS*

Style Property in Java FX

- In JavaFX, a style property is defined with a prefix **–fx-**
- Each node has its own style properties
- You can find these properties from
- http://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html
- The syntax for setting a style is **styleName:value**

Style Property in Java FX

- Multiple style properties for a node can be set together separated by semicolon (;)

- Example: circle.setStyle("**-fx-stroke: black; -fx-fill: red;**");
  - sets two JavaFX CSS properties for a circle

- Equivalent statements:
  - circle.setStroke(Color.BLACK);
  - circle.setFill(Color.RED);

Stroke

- To set style color in JavaFX:

- Use -fx-stroke for coloring lines (using CSS)
  - line.setStyle("-fx-stroke: red;");

Or

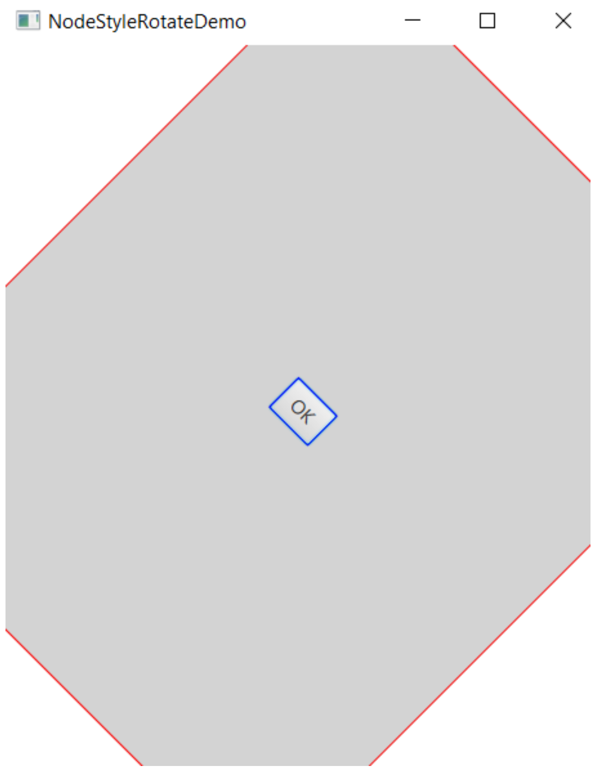
- call setStroke() for coloring lines (using Java API)
  - line.setStroke(Color.RED);

Style Property in Java FX

- If an incorrect JavaFX CSS is used, your program will still compile and run, but the style is ignored.

Example 07

- Let's look at the example



NodeStyleRotateDemo.java

NodeStyleRotateDemo-Explanation

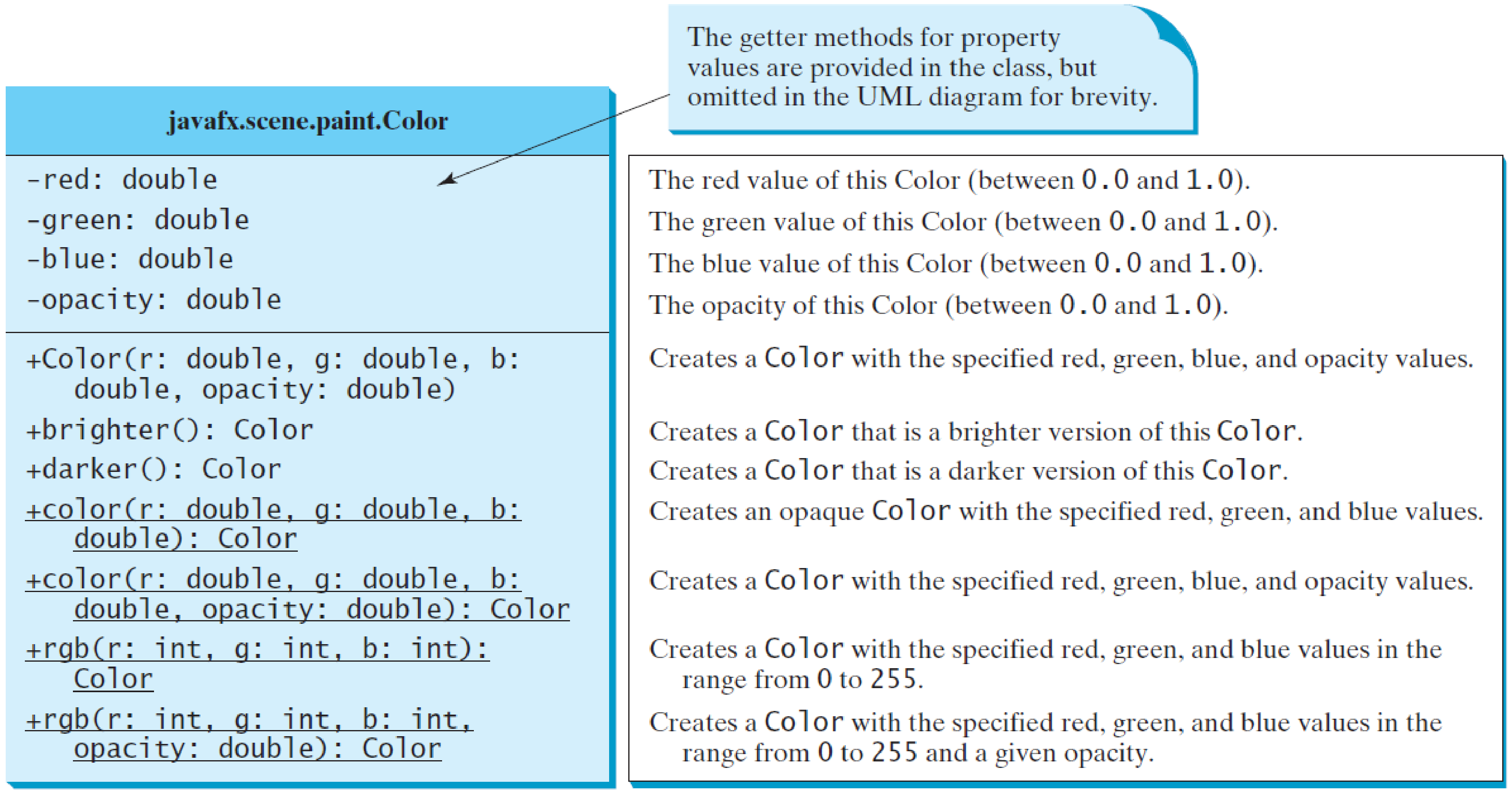
- The rotate on a pane causes all its containing nodes rotated too.
- The **Node** class contains many useful methods that can be applied to all nodes.

The Color Class

- *The* **Color** *class can be used to create colors.*

- JavaFX defines the abstract **Paint** class for painting a node. The **javafx.scene.paint.Color** is a concrete subclass of **Paint**, which is used to encapsulate colors,

The Color Class(Cont.)



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

javafx.scene.paint.Color

-red: double	The red value of this Color (between 0.0 and 1.0).
-green: double	The green value of this Color (between 0.0 and 1.0).
-blue: double	The blue value of this Color (between 0.0 and 1.0).
-opacity: double	The opacity of this Color (between 0.0 and 1.0).
+Color(r: double, g: double, b: double, opacity: double)	Creates a Color with the specified red, green, blue, and opacity values.
+brighter(): Color	Creates a Color that is a brighter version of this Color.
+darker(): Color	Creates a Color that is a darker version of this Color.
+color(r: double, g: double, b: double): Color	Creates an opaque Color with the specified red, green, and blue values.
+color(r: double, g: double, b: double, opacity: double): Color	Creates a Color with the specified red, green, blue, and opacity values.
+rgb(r: int, g: int, b: int): Color	Creates a Color with the specified red, green, and blue values in the range from 0 to 255.
+rgb(r: int, g: int, b: int, opacity: double): Color	Creates a Color with the specified red, green, and blue values in the range from 0 to 255 and a given opacity.

The **Font** Class

- *A* **Font** *describes font name, weight, and size.*
- You can set fonts for rendering the text
- The **javafx.scene.text.Font** class is used to create fonts
- A **Font** instance can be constructed using its constructors or using its static methods.
- **Font** is defined by its name, weight, posture, and size.
- Times, Courier, and Arial are the examples of the font names.

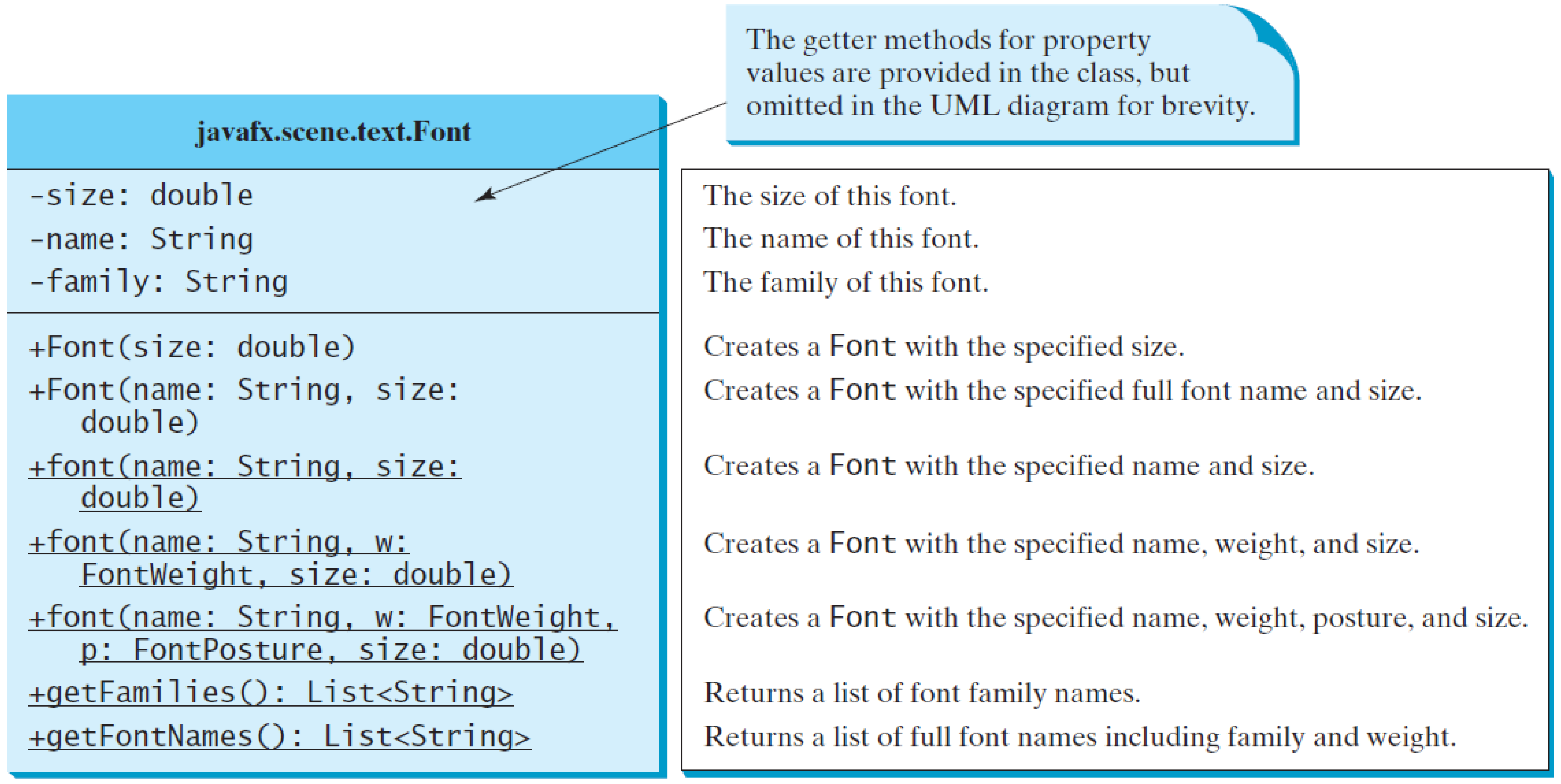
The **Font** Class(Cont.)

- You can obtain a list of available font family names by invoking the static **getFamilies()** method.

Example of creating fonts:

- Font font1 = **new** Font("**SansSerif**", **16**);
- Font font2 = Font.font("**Times New Roman**", FontWeight.BOLD, FontPosture.ITALIC, **12**);

The **Font** Class(Cont.)

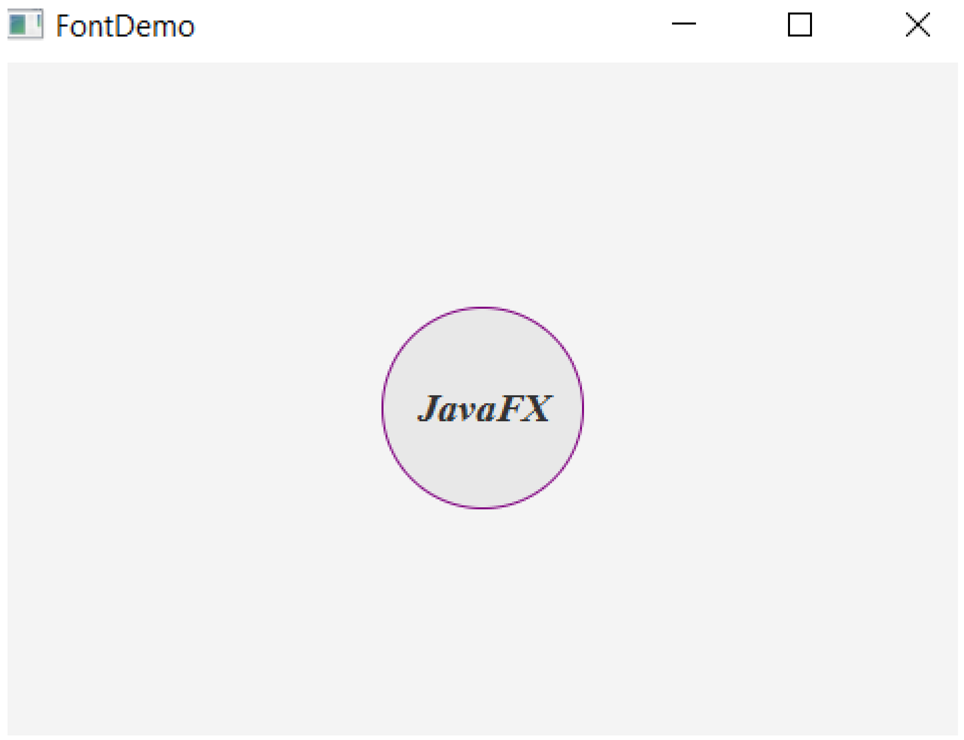


The getter methods for property values are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.text.Font

`-size: double`	The size of this font.
`-name: String`	The name of this font.
`-family: String`	The family of this font.
`+Font(size: double)`	Creates a `Font` with the specified size.
`+Font(name: String, size: double)`	Creates a `Font` with the specified full font name and size.
`+font(name: String, size: double)`	Creates a `Font` with the specified name and size.
`+font(name: String, w: FontWeight, size: double)`	Creates a `Font` with the specified name, weight, and size.
`+font(name: String, w: FontWeight, p: FontPosture, size: double)`	Creates a `Font` with the specified name, weight, posture, and size.
`+getFamilies(): List<String>`	Returns a list of font family names.
`+getFontNames(): List<String>`	Returns a list of full font names including family and weight.

Example08 - `FontDemo.java`



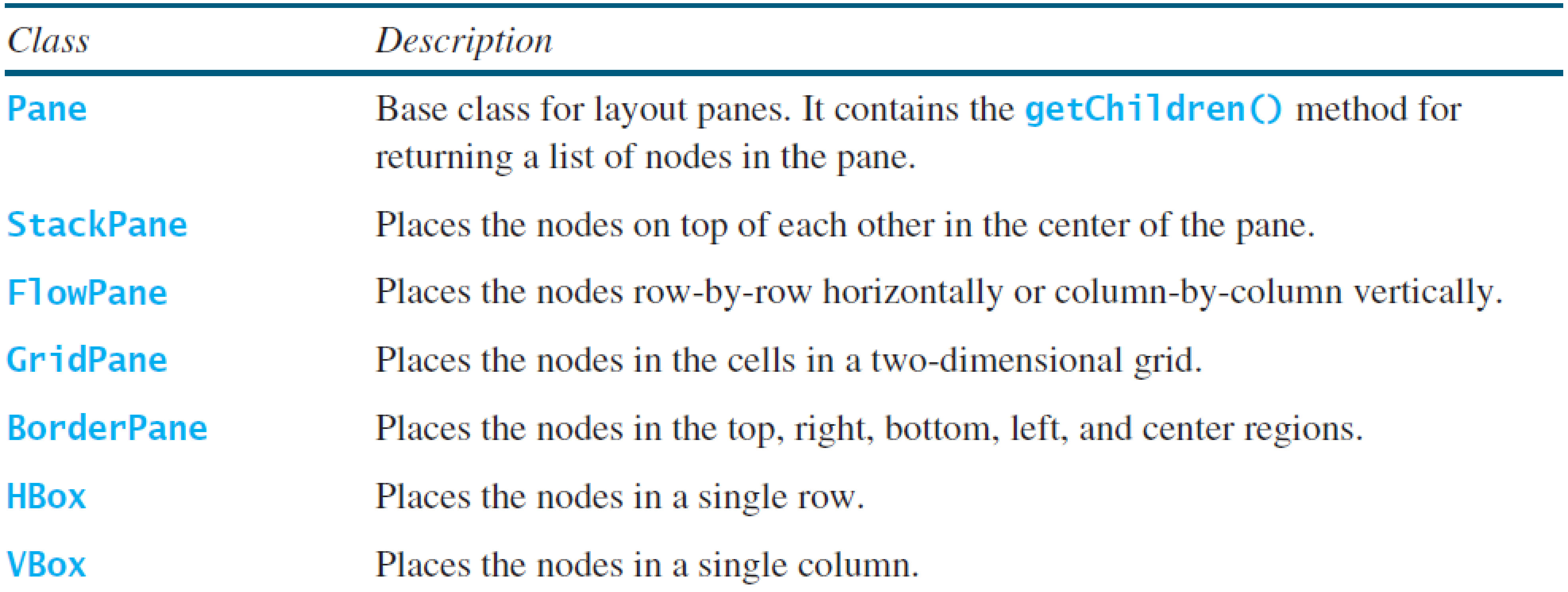
A label is on top of a circle displayed in the center of the scene

Explanation – Example08

- A **StackPane** places the nodes in the center and nodes are placed on top of each other.
- A custom color is created and set as a fill color for the circle
- As you resize the window, the circle and label are displayed in the center of the window, because the circle and label are placed in the stack pane. Stack pane automatically places nodes in the center of the pane.

Layout Panes

- ➢ *JavaFX provides many types of panes for automatically laying out nodes in a desired location and size.`*

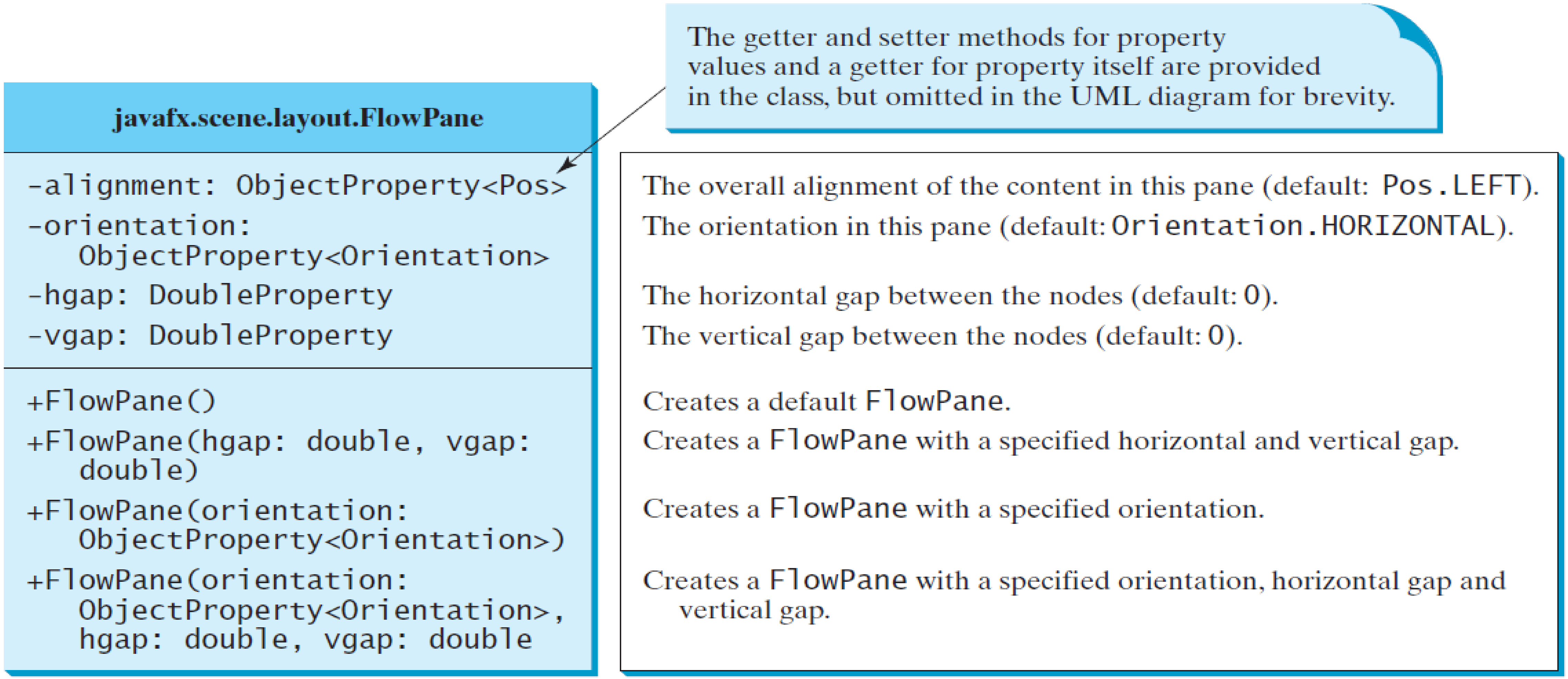


Class	*Description*
`Pane`	Base class for layout panes. It contains the `getChildren()` method for returning a list of nodes in the pane.
`StackPane`	Places the nodes on top of each other in the center of the pane.
`FlowPane`	Places the nodes row-by-row horizontally or column-by-column vertically.
`GridPane`	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** arranges the nodes in the pane horizontally from left to right or vertically from top to bottom in the order in which they were added.

FlowPane

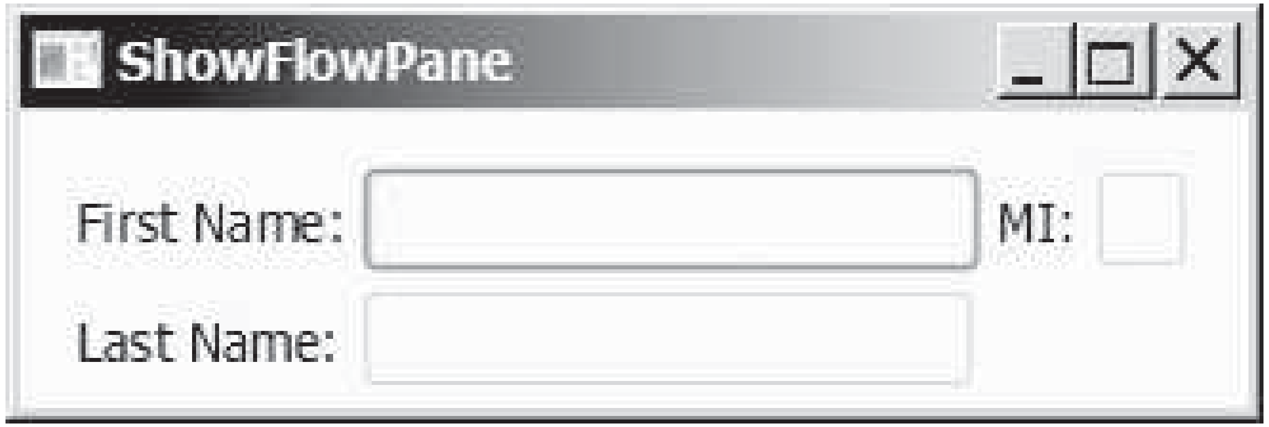


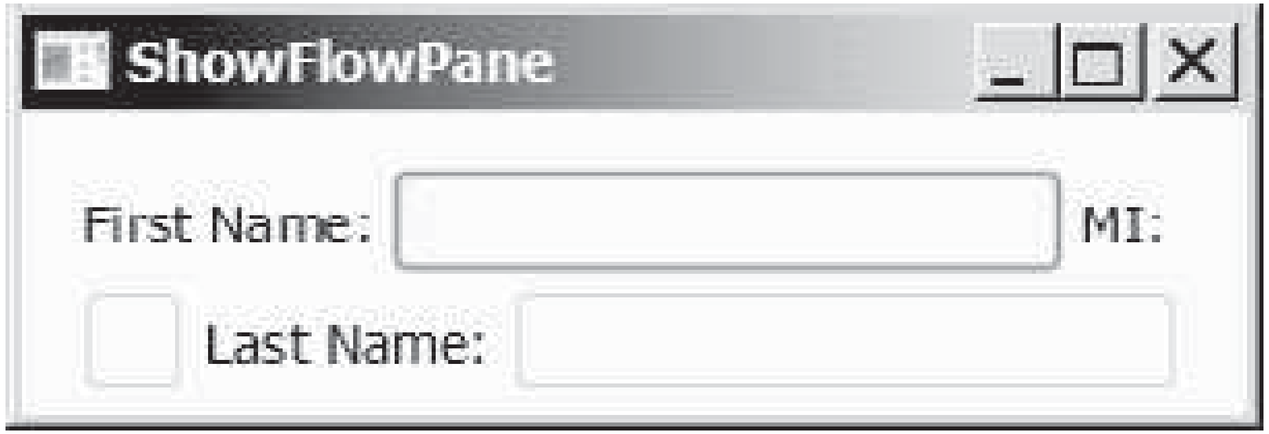
The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.layout.FlowPane

`-alignment: ObjectProperty<Pos>`	The overall alignment of the content in this pane (default: `Pos.LEFT`).
`-orientation: ObjectProperty<Orientation>`	The orientation in this pane (default: `Orientation.HORIZONTAL`).
`-hgap: DoubleProperty`	The horizontal gap between the nodes (default: `0`).
`-vgap: DoubleProperty`	The vertical gap between the nodes (default: `0`).
`+FlowPane()`	Creates a default `FlowPane`.
`+FlowPane(hgap: double, vgap: double)`	Creates a `FlowPane` with a specified horizontal and vertical gap.
`+FlowPane(orientation: ObjectProperty<Orientation>)`	Creates a `FlowPane` with a specified orientation.
`+FlowPane(orientation: ObjectProperty<Orientation>, hgap: double, vgap: double`	Creates a `FlowPane` with a specified orientation, horizontal gap and vertical gap.

FlowPane(Example)





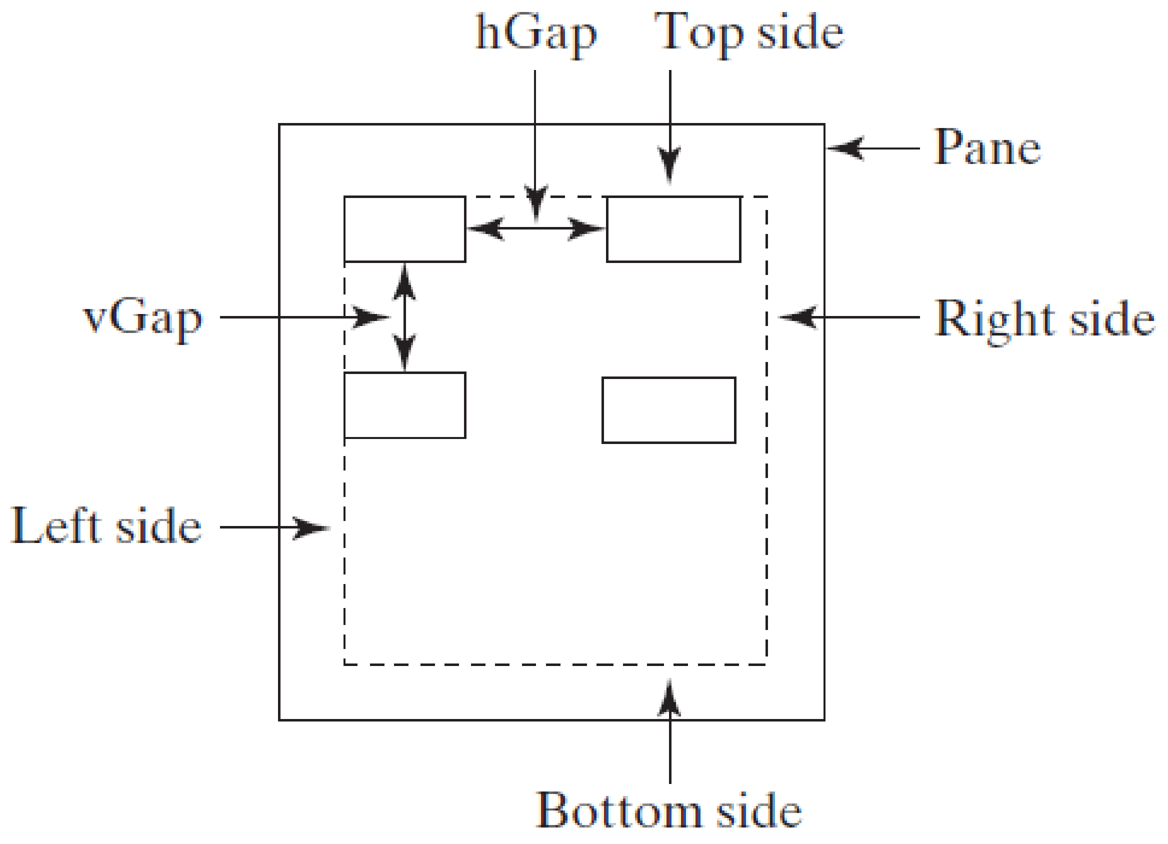
FlowPane(Example)

- Invoking **tfMi.setPrefColumnCount(1)** sets the preferred column count to **1** for the MI text field

FlowPane(Example)

- The program creates a **FlowPane** and sets its **padding** property with an **Insets** object
- An **Insets** object specifies the size of the border of a pane
- The constructor **Insets(11, 12, 13, 14)** creates an **Insets** with the border sizes for top (11), right (12), bottom (13), and left (14) in pixels
- The **hGap** and**vGap** properties are in lines 15–16 to specify the horizontal gap and vertical gap between two nodes in the pane

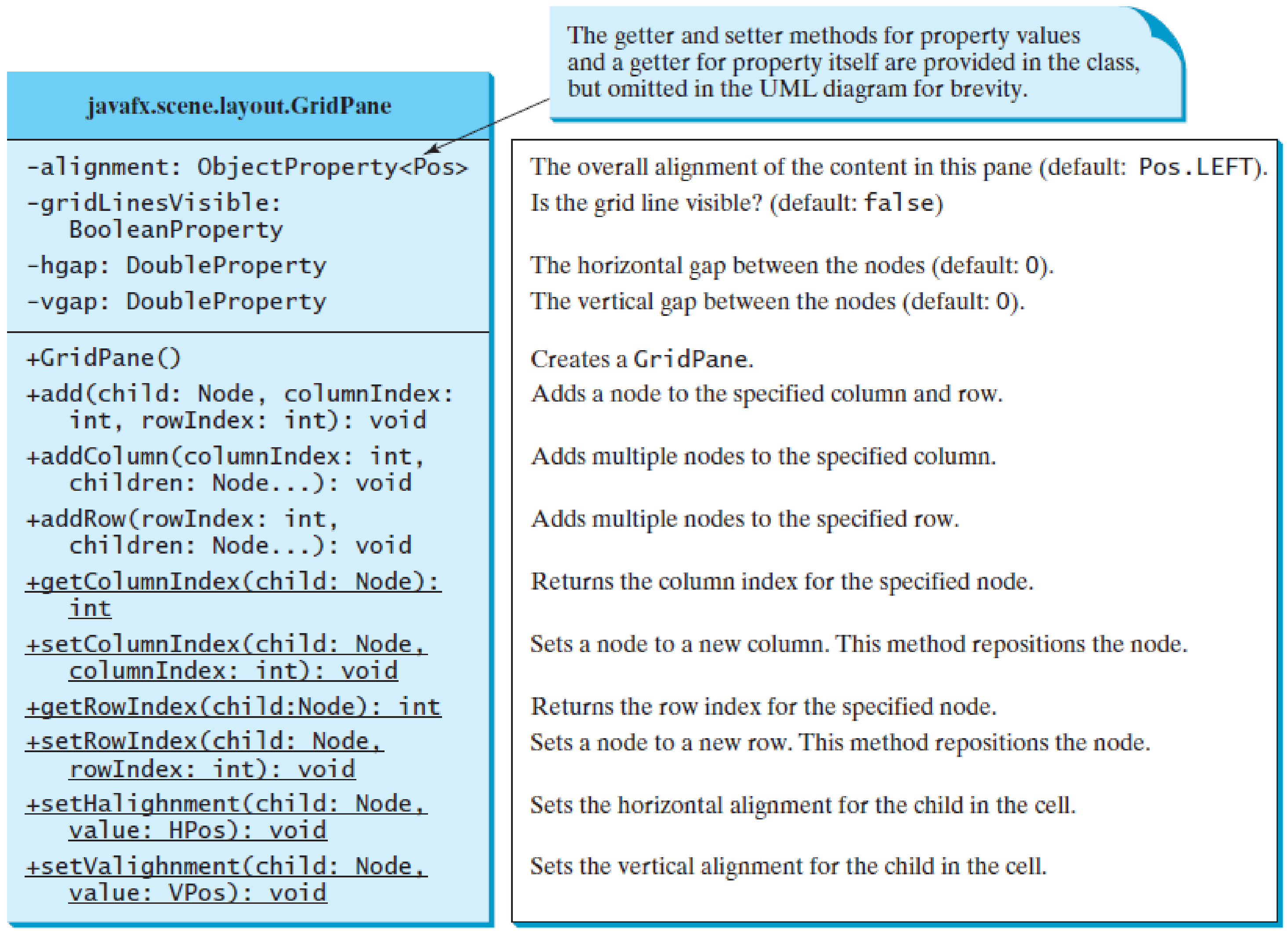
FlowPane(Example)



GridPane

- A **GridPane** arranges nodes in a grid (matrix) formation

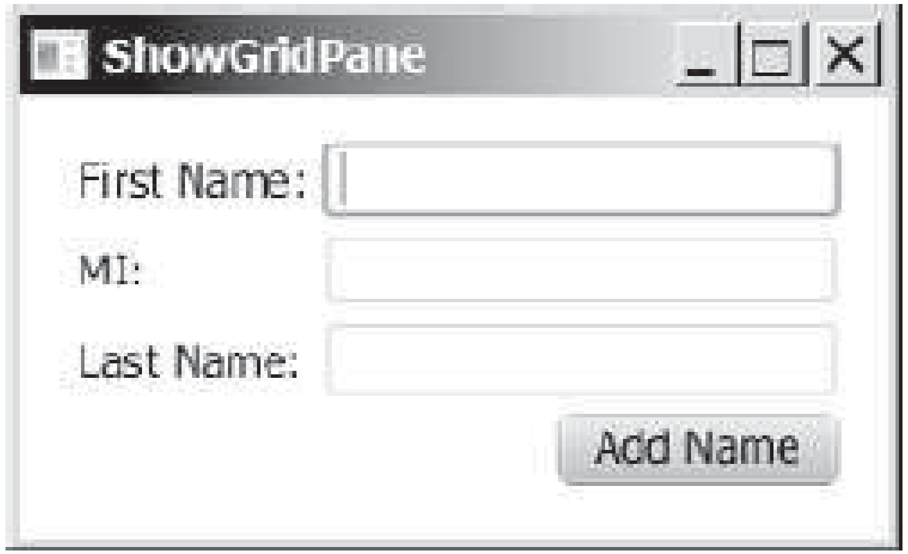
GridPane

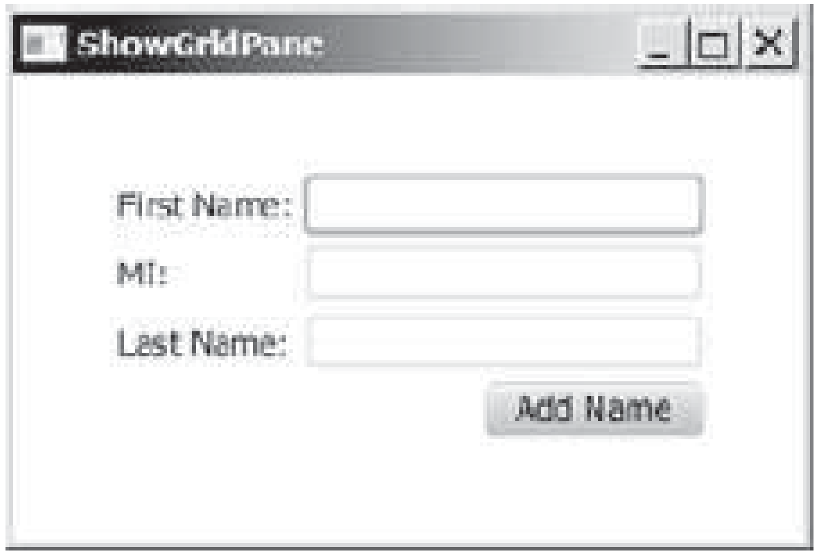


The getter and setter methods for property values and a getter for property itself are provided in the class, but omitted in the UML diagram for brevity.

javafx.scene.layout.GridPane	
`-alignment: ObjectProperty<Pos>`	The overall alignment of the content in this pane (default: `Pos.LEFT`).
`-gridLinesVisible: BooleanProperty`	Is the grid line visible? (default: `false`)
`-hgap: DoubleProperty`	The horizontal gap between the nodes (default: 0).
`-vgap: DoubleProperty`	The vertical gap between the nodes (default: 0).
`+GridPane()`	Creates a `GridPane`.
`+add(child: Node, columnIndex: int, rowIndex: int): void`	Adds a node to the specified column and row.
`+addColumn(columnIndex: int, children: Node...): void`	Adds multiple nodes to the specified column.
`+addRow(rowIndex: int, children: Node...): void`	Adds multiple nodes to the specified row.
`+getColumnIndex(child: Node): int`	Returns the column index for the specified node.
`+setColumnIndex(child: Node, columnIndex: int): void`	Sets a node to a new column. This method repositions the node.
`+getRowIndex(child:Node): int`	Returns the row index for the specified node.
`+setRowIndex(child: Node, rowIndex: int): void`	Sets a node to a new row. This method repositions the node.
`+setHalighnment(child: Node, value: HPos): void`	Sets the horizontal alignment for the child in the cell.
`+setValighnment(child: Node, value: VPos): void`	Sets the vertical alignment for the child in the cell.

ShowGridPane.Java


ShowGridPane
First Name:
MI:
Last Name:
Add Name


ShowGridPane
First Name:
MI:
Last Name:
Add Name