**CHAPETR 14: JAVAFX BASICS**

* 1. **INTRODUCTION**

JavaFX is an excellent pedagogical tool for learning object-oriented programming.

JavaFX is a new framework for developing Java GUI programs.

The JavaFX API is an excellent example of how the object-oriented principles are applied.

* 1. **JavaFX VS Swing and AWT**

Swing and AWT are replaced by the JavaFX platform for developing rich GUI applications.

When Java was introduced, the GUI classes were bundled in a library known as the Abstract Windows Toolkit (AWT). 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. Swing components are painted directly on canvases using Java code. Swing components depend less on the target platform, and use less of the native GUI resources.

Swing is designed for developing desktop GUI applications. It is now replaced by a completely new GUI platform known as JavaFX. JavaFX incorporates modern GUI technologies to enable you to develop rich GUI applications.

JavaFX pro vides a multitouch support for touch-enabled devices such as tablets and smart phones. JavaFX has a built-in 2D, 3D, animation support, and video and audio playback. Using third party software, you can develop JavaFX programs to be deployed on devices running iOS or Android.

* 1. **THE BASIC STRUCTURE OF A JavaFX Program**

The **javafx.application.Application** class defines the essential framework for writing JavaFX programs.

We begin by writing a simple JavaFX program that illustrates the basic structure of a JavaFX program. Every JavaFX program is defined in a class that extends **javafx.application.Application**

The following **MyjavaFX** program shows the javaFX structure:

**import javafx.application.Application;**

**import javafx.scene.Scene;**

**import javafx.scene.control.Button;**

**import javafx.stage.Stage;**

**public class MyJavaFX extends Application {**

**@Override // Override the start method in the Application class**

**public void start(Stage primaryStage) {**

**// Create a scene and place a button 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) {**

**Application.launch(args);**

**}**

**}**

* 1. **Panes, Groups, UI Controls, and Shapes**

Panes, Groups, UI controls, and shapes are subtypes of Node.

When you run MyJavaFX, 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.

You place nodes inside a pane then place the pane into a scene.

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, arc, polygon, polyline, and so on.

A UI control refers to a label, button, check box, radio button, text field, text area, and so on. A group is a container that groups a collection of nodes.

* 1. **PROPERTY BINDING**

You can bind a target object to a source object. A change in the source object will be automatically reflected in the target object.

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 objects changes, the target object is also automatically changed.

The target object is called a binding object or a binding property, and the source object is called a bindable object or observable object.

A binding property is an object that can be bound to a source object.

A target listens to the changes in the source and automatically updates itself once a change is made in the source.

A target binds with a source using the bind method as follows:

**target.bind(source);**

The **bind** method is defined in the **javafx.beans.property.Property** interface. A binding property is an instance of **javafx.beans.property.Property**. An observable source object is an instance of the **javafx.beans.value.ObservableValue** interface.

An **ObservableValue** is an entity that wraps a value and allows to observe the value for changes.

A binding property is an object. JavaFX defines binding properties for primitive types and strings. For a **double/float/long/int/boolean** value, its binding property type is **DoubleProperty/FloatProperty/LongProperty/IntegerProperty/BooleanProperty**, respectively.

For a string, its binding property type is **StringProperty**.

* 1. **COMMON PROPERTIES AND METHODS FOR Nodes**

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

Nodes share many common properties. This section introduces two such properties: style and rotate.

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. In JavaFX, a style property is defined with a prefix **–fx–**. Each node has its own style properties.

The syntax for setting a style is **styleName:value**. Multiple style properties for a node can be set together separated by semicolon (;).

For example, the following statement sets two JavaFX CSS properties for a circle:

**circle.setStyle("−fx−stroke: black; −fx−fill: red;");**

This statement is equivalent to the following two statements:

**circle.setStroke(Color.BLACK);**

**circle.setFill(Color.RED);**

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

* 1. **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.

|  |  |
| --- | --- |
| 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. |

Color encapsulates information about colors.

A color instance can be constructed using the following constructor:

**public Color(double r, double g, double b, double opacity);**

In which r, g, and b specify a color by its red, green, and blue components with values in the range from 0.0 (darkest shade) to 1.0 (lightest shade).

You can use one of the many standard colors such as **BEIGE, BLACK, BLUE, BROWN, CYAN, DARKGRAY, GOLD, GRAY, GREEN, LIGHTGRAY, MAGENTA, NAVY, ORANGE, PINK, RED, SILVER, WHITE, and YELLOW** defined as constants in the Color class.

* 1. **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.

|  |  |
| --- | --- |
| 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. |
| +getFontNames(): List | Returns a list of all font names installed on the user system. |

**Font** encapsulates information about fonts.

* 1. **THE Image and ImageView Classes**

The **Image** class represents a graphical image, and the **ImageView** class can be used to display an image.

The **javafx.scene.image.Image** class represents a graphical image and is used for loading an image from a specified filename or a URL.

For example, the following code creates an ImageView from an image file:

**Image image = new Image("image/us.gif");**

**ImageView imageView = new ImageView(image);**

Alternatively, you can create an ImageView directly from a file or a URL as follows:

**ImageView imageView = new ImageView("image/us.gif");**

|  |  |
| --- | --- |
| javafx.scene.image.Image | |
| –error: ReadOnlyBooleanProperty | Indicates whether the image is loaded correctly? |
| –height: ReadOnlyDoubleProperty | The height of the image. |
| –width: ReadOnlyDoubleProperty | The width of the image. |
| –progress: ReadOnlyDoubleProperty | The approximate percentage of image’s loading that is completed. |
| +Image(filenameOrURL: String) | Creates an Image with contents loaded from a file or a URL. |

**Image** encapsulates information about images.

|  |  |
| --- | --- |
| javafx.scene.image.ImageView | |
| –f itHeight: DoubleProperty | The height of the bounding box within which the image is resized to fit. |
| –f itWidth: DoubleProperty | The width of the bounding box within which the image is resized to fit. |
| –x: DoubleProperty | The x-coordinate of the ImageView origin. |
| –y: DoubleProperty | The y-coordinate of the ImageView origin. |
| –image: ObjectProperty | The image to be displayed in the image view. |
| +ImageView() | Creates an ImageView. |
| +ImageView(image: Image) | Creates an ImageView with the specified image. |
| +ImageView(filenameOrURL: String) | Creates an ImageView with image loaded from the specified file or URL. |

**ImageView** is a node for displaying an image.

* 1. **Layout Panes and Groups**

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

Panes and groups are the containers for holding nodes. The **Group** class is often used to group nodes and to perform transformation and scale as a group. Panes and UI control objects are resizable, but group, shape, and text objects are not resizable. JavaFX provides many types of panes for organizing nodes in a container:

|  |  |
| --- | --- |
| 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. |

* + - 1. **FlowPlane**

**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.

When one row or one column is filled, a new row or column is started.

You can specify the way the nodes are placed horizontally or vertically using one of two constants: **Orientation.HORIZONTAL** or **Orientation.VERTICAL**.

|  |  |
| --- | --- |
| javafx.scene.layout.FlowPane | |
| –alignment: ObjectProperty | The overall alignment of the content in this pane (default: Pos.LEFT). |
| –orientation: ObjectProperty | 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) | Creates a FlowPane with a specified orientation. |
| +FlowPane(orientation: ObjectProperty, hgap: double, vgap: double) | Creates a FlowPane with a specified orientation, horizontal gap and vertical gap. |

FlowPane lays out nodes row-by-row horizontally or column-by-column vertically.

* + 1. **GridPane**

A **GridPane** arranges nodes in a grid (matrix) formation. The nodes are placed in the specified column and row indices.

The class diagram for **GridPane** is shown below:

|  |  |
| --- | --- |
| javafx.scene.layout.GridPane | |
| –alignment: ObjectProperty  –gridLinesVisible: BooleanProperty  –hgap: DoubleProperty  –vgap: DoubleProperty | The overall alignment of the content in this pane (default: Pos.LEFT).  Is the grid line visible? (default: false)  The horizontal gap between the nodes (default: 0).  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. |

**GridPane** lays out nodes in the specified cell in a grid.

* + 1. **BorderPane**

A **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) methods**.

The class diagram for **BorderPane** is shown below:

|  |  |
| --- | --- |
| javafx.scene.layout.BorderPane | |
| –top: ObjectProperty | The node placed in the top region (default: null). |
| –right: ObjectProperty | The node placed in the right region (default: null). |
| –bottom: ObjectProperty | The node placed in the bottom region (default: null). |
| –left: ObjectProperty | The node placed in the left region (default: null). |
| –center: ObjectProperty | The node placed in the center region (default: null). |
| +BorderPane() | Creates a BorderPane. |
| +BorderPane(node: Node) | Creates a BorderPane with the node placed in the center of the plane |
| +setAlignment(child: Node, pos: Pos) | Sets the alignment of the node in the BorderPane. |

**BorderPane** places the nodes in top, bottom, left, right, and center regions.

* + 1. **HBox and VBox**

An **HBox** lays out its children in a single horizontal row.

A **VBox** lays out its children in a single vertical column.

Recall that a **FlowPane** can lay out its children in multiple rows or multiple columns, but an **HBox** or a **VBox** can lay out children only in one row or one column. The class diagrams for **HBox** and **VBox** are shown below:

|  |  |
| --- | --- |
| javafx.scene.layout.HBox | |
| –alignment: ObjectProperty | The overall alignment of the children in the box (default: Pos.TOP\_LEFT). |
| –fillHeight: BooleanProperty | Is resizable children fill the full height of the box (default: true). |
| –spacing: DoubleProperty | The horizontal gap between two nodes (default: 0). |
| +HBox() | Creates a default HBox |
| +HBox(spacing: double) | Creates an HBox with the specified horizontal gap between nodes. |
| +setMargin(node: Node, value: Insets): void | Sets the margin for the node in the pane. |

**HBox** places the nodes in one row.

|  |  |
| --- | --- |
| javafx.scene.layout.VBox | |
| –alignment: ObjectProperty | The overall alignment of the children in the box (default: Pos.TOP\_LEFT). |
| –fillWidth: BooleanProperty | Is resizable children fill the full width of the box (default: true). |
| –spacing: DoubleProperty | The vertical gap between two nodes (default: 0). |
| +VBox() | Creates a default VBox. |
| +VBox(spacing: double) | Creates a VBox with the specified horizontal gap between nodes. |
| +setMargin(node: Node, value: Insets): void | Sets the margin for the node in the pane. |

**VBox** places the nodes in one column.

* 1. **SHAPES**

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

The **Shape** class is the abstract base class that defines the common properties for all shapes. Among them are the **fill**, **stroke**, and **strokeWidth** properties.

The **fill** property specifies a color that fills the interior of a shape.

The **stroke** property specifies a color that is used to draw the outline of a shape. The **strokeWidth** property specifies the width of the outline of a shape.

This section introduces the classes **Text**, **Line**, **Rectangle**, **Circle**, **Ellipse**, **Arc**, **Polygon**, and **Polyline** for drawing texts and simple shapes.

All these are subclasses of Shape, as shown below:

Node

Shape

Polygon

Arc

Ellipse

Circle

Rectangle

Line

Text

Polyline

A shape is a node. The **Shape** class is the root of all shape classes.

* **Text:** The **Text** class defines a node that displays a string at a starting point (**x, y**).

A **Text** object is usually placed in a pane. The pane’s upper-left corner point is (0, 0) and the bottom-right point is (**pane.getWidth(), pane.getHeight()).** A string may be displayed in multiple lines separated by \n.

* **Line:** A line connects two points with four parameters **startX**, **startY**, **endX**, and **endY**.

The **Line** class defines a line.

* **Rectangle:** A rectangle is defined by the parameters x, y, width, height, **arcWidth**, and **arcHeight**.

The rectangle’s upper-left corner point is at (**x, y**), parameter **aw** (**arcWidth**) is the horizontal diameter of the arcs at the corner, and **ah** (**arcHeight**) is the vertical diameter of the arcs at the corner.

The **Rectangle** class defines a rectangle.

* **Circle and Ellipse**: A circle is defined by its parameters **centerX**, **centerY**, and **radius**. The **Circle** class defines a circle.

An ellipse is defined by its parameters **centerX**, **centerY**, **radiusX**, and **radius**.

The **Ellipse** class defines an ellipse.

* **Arc**: An arc is conceived as part of an ellipse, defined by the parameters **centerX, centerY, radiusX, radiusY, startAngle**, length, and an arc type (**ArcType.OPEN, ArcType .CHORD, or ArcType.ROUND**).

The parameter **startAngle** is the starting angle, and length is the spanning angle.

Angles are measured in degrees and follow the usual mathematical conventions.

The **Arc** class defines an arc.

* **Polygon and Polyline**: The **Polygon** class defines a polygon that connects a sequence of points.

The **Polyline** class is similar to the **Polygon** class except that the **Polyline** class is not automatically closed.

The **Polygon** class defines a polygon.

**THE END!**