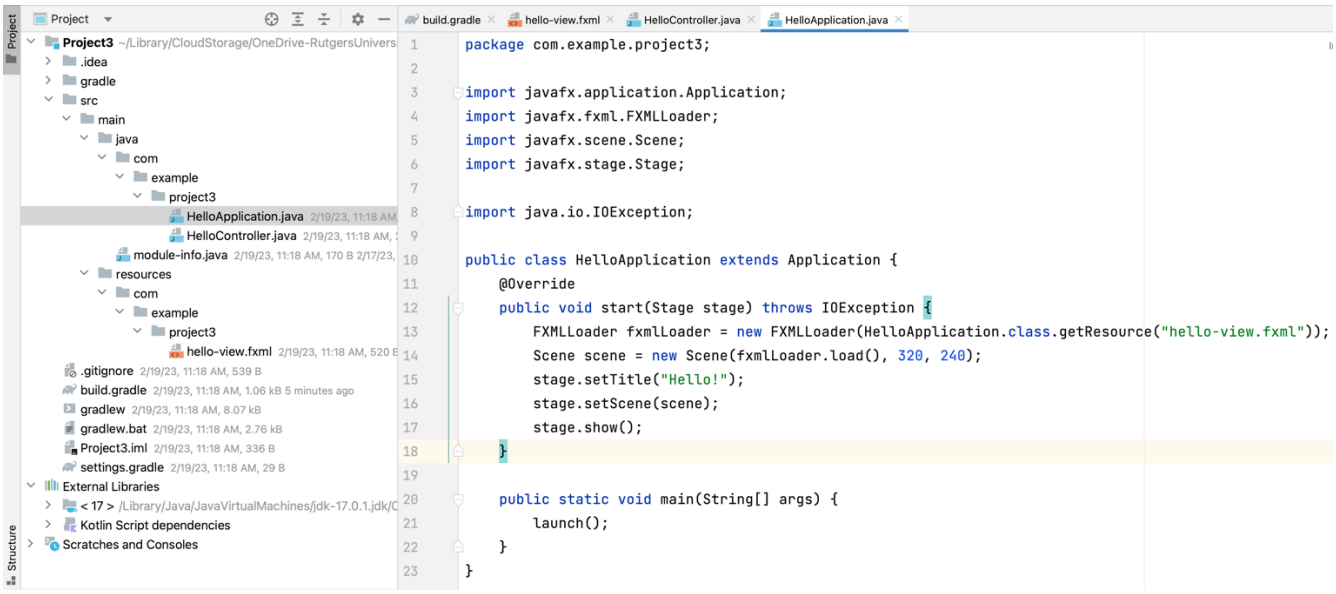
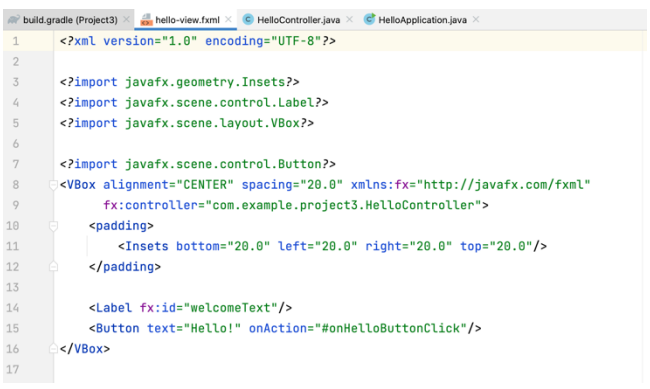
javafx.application.Application class defines the essential framework for writing JavaFX programs. After you created a JavaFX project in IntelliJ, 3 files will be automatically generated:

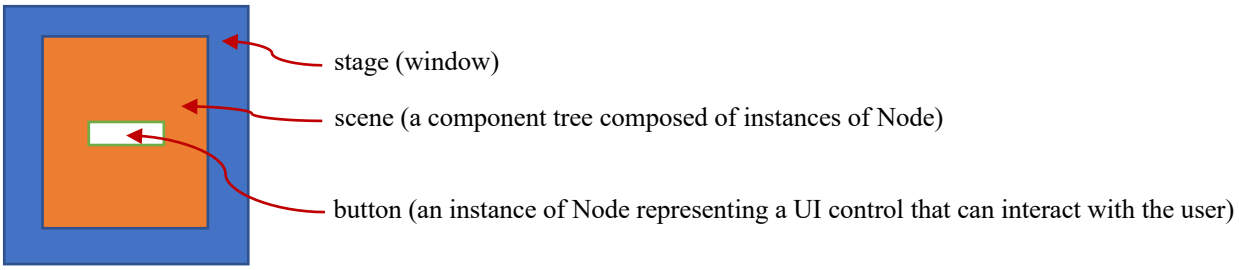
1. **HelloApplicatoin.java**, this is the main class that contains the main() method to "launch" the application.
2. **HelloController.java**, this is the "controller" class that contains the "event-handlers" to handle the events triggered on the GUI (Graphical User Interface.)
3. **hello-view.fxml**, this is the "view" class which define and layout the UI component objects on the screen. The layout of the UI is coded with the eXtensible Markup Language (XML), with special tags for JavaFX. For example, **fx:controller** defines the controller class associated with the "view" to handle the events.





The **launch()** method is a static method defined in the **Application** class for launching a stand-alone JavaFX application. The **main()** method is not needed if you run the program from the command line. It may be needed to launch a JavaFX program from an IDE with a limited JavaFX support. When you run a JavaFX application without a main method, JVM automatically invokes the **launch()** method to run the application.

The **HelloApplication class in IntellJ** 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 default constructor and invokes its start() method. The start() method normally places UI (User Interface) control objects in a scene and displays the scene in a stage, as shown below. **Stage** is a window for displaying a scene that contains nodes. A node represents an object on the UI, such as a button a user can click. Multiple stages (multiple windows) can be displayed in a JavaFX program.



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.

A **Stage** object is a window. A **Stage** object called **stage** is automatically created by the JVM when the application is launched. It is the parameter of the start() method. The .setScene() method sets the scene to the stage and the .show() method displays the stage on the screen. JavaFX names the **Stage** and **Scene** classes using the analogy from the theater. You may think of Stage as the platform to support scenes, and **nodes** as actors to perform in the scenes. You can create additional stages if needed. For example,

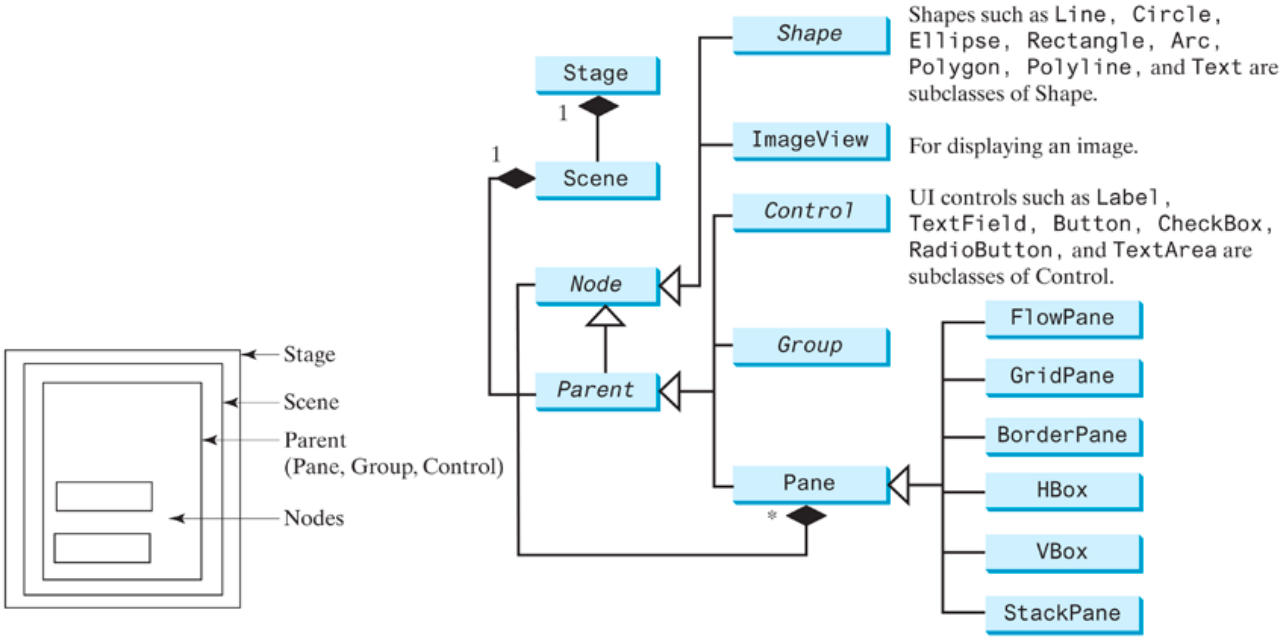
```
public void newStage(ActionEvent actionEvent) throws IOException {
    //second stage
    Parent root = FXMLLoader.load(getClass().getResource( name: "newStage.fxml"));
    Stage stage = new Stage(); // Create a new stage
    stage.setTitle("Second Stage"); // Set the stage title
    stage.setScene(new Scene(root, v: 300,  v1: 275));
    stage.show(); // Display the stage
}
```

By default, the user can resize the stage. To prevent the user from resizing the stage, invoke **stage.setResizable(false)**.

- **Panes, Groups, UI Controls, and Shapes**

When you run a JavaFX application, the window is displayed as shown on the left. The button can be displayed on the window by setting the position and size properties of a button. However, a better approach is to use the 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. You can apply transformations or effects to a group, which automatically apply to all the children in the group. A **scene** can be displayed in a stage. The relationship

among Stage, Scene, Node, Control, Group, and Pane is illustrated in the Class diagram below. Note 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. You can create a Scene using the constructor Scene(Parent, width, height) or Scene(Parent). The dimension of the scene is automatically decided in the latter constructor. Every subclass of Node has a default constructor for creating a default node.



- **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. The target object is called a binding object or a **binding property**, and the source object is called a bindable object or **observable object**. For example, to always show a Circle object on the center of the window whenever the window is resized, 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, as the code given below.

```
Pane pane = new Pane();
Circle circle = new Circle();
circle.centerXProperty().bind(pane.widthProperty().divide(2));
circle.centerYProperty().bind(pane.heightProperty().divide(2));
...
```

The Circle class has the centerX property for representing the x-coordinate of the circle center. This property like many properties in JavaFX classes can be used both as target and source in a property binding. 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.

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. These properties are also subtypes of ObservableValue. Therefore, they can be used as both source and target in a binding.

By convention, each binding property (e.g., centerX) in a JavaFX class (e.g., Circle) has a getter (e.g., getCenterX()) and setter (e.g., setCenterX(double)) method for returning and setting the property's value. It also has a getter method for returning the property itself. The naming convention for this method is the property name followed by the word Property. For example, the property getter method for centerX is centerXProperty(). We call the getCenterX() method as the value getter method, the setCenterX(double) method as the value setter method, and centerXProperty() as the property getter method. Note getCenterX() returns a double value, and centerXProperty() returns an object of the DoubleProperty type.

- **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. For example, **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. See the JavaFX CSS Reference Guide: https://openjfx.io/javadoc/17/javafx.graphics/javafx/scene/doc-files/cssref.html

The rotate property enables you to specify an angle in degrees for rotating a node from its center. If the degree is positive, the rotation is performed clockwise; otherwise, it is performed counterclockwise. For example, the following code rotates a button 80 degrees: **button.setRotate(80);**

The Node class contains many useful methods that can be applied to all nodes. For example, you can use the **contains(double x, double y)** method to test whether a point (x, y) is inside the boundary of a node and use the **setScaleX(double scale**) and **setScaleY(double scale)** methods to scale a node.

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, as shown below.

A color instance can be constructed using the following constructor:

Color(double red, double green, double blue, double opacity)

where red, green and blue 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). The opacity value defines the transparency of a color within the range from 0.0 (completely transparent) to 1.0 (completely opaque). This is known as the RGBA model, where RGBA stands for red, green, blue, and alpha. The alpha value indicates the opacity.

The Color class is immutable. Once a Color object is created, its properties cannot be changed. The brighter() method returns a new Color with a larger red, green, and blue values, and the darker() method returns a new Color with a smaller red, green, and blue values. The opacity value is the same as in the original Color object. Some examples are as follows. For more details, see: https://openjfx.io/javadoc/17/javafx.graphics/javafx/scene/paint/Color.html

```
Color c = Color.BLUE;   //use the blue constant
Color c = new Color(0,0,1,1.0);
//standard constructor, use 0->1.0 values, explicit alpha of 1.0
Color c = Color.color(0,0,1.0); //use 0->1.0 values. implicit alpha of 1.0
```

```
Color c = Color.color(0,0,1.0,1.0); //use 0->1.0 values, explicit alpha of 1.0
Color c = Color.rgb(0,0,255); //use 0->255 integers, implicit alpha of 1.0
Color c = Color.rgb(0,0,255,1.0); //use 0->255 integers, explicit alpha of 1.
```

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. A Font is defined by its name, weight, posture, and size. Times New Roman, Courier, and Arial are examples of font names. You can obtain a list of available font family names by invoking the **static getFontNames()** method. This method returns List. **List** is an interface that defines common methods for lists. **ArrayList** is a concrete class that implements List. For example,

```
Pane pane = new StackPane();
Circle circle = new Circle();
circle.setRadius(50);
circle.setStroke(Color.BLACK);
circle.setFill(new Color(0.5,0.5, 0.5, 0.1));
pane.getChildren().add(circle);
Label label = new Label("JavaFX");
label.setFont(Font.font("Times New Roman", FontWeight.BOLD, FontPosture.ITALIC,20));
...
```

For more details, see: https://openjfx.io/javadoc/17/javafx.graphics/javafx/scene/text/Font.html

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, **new Image("image/us.gif")** creates an Image object for the image file us.gif under the directory image in the Java class directory and **new Image("http://liveexample.pearsoncmg.com/book/image/us.gif")** creates an Image object for the image file in the URL on the Web.

The **javafx.scene.image.ImageView** is a node for displaying an image. An ImageView can be created from an Image object. For example, the following code creates an ImageView from an image file:

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

For more information, see:
https://openjfx.io/javadoc/17/javafx.graphics/javafx/scene/image/Image.html

- **Layout Panes**

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. Below is the inheritance tree structure for the layout Panes in JavaFX Javadoc. SceneBuilder provides additional layout panes. Please refer to the UI components under "container" in the SceneBuilder.

- javafx.scene.layout.**Pane**
  - javafx.scene.layout.**AnchorPane**
  - javafx.scene.layout.**BorderPane**
  - javafx.scene.layout.**FlowPane**
  - javafx.scene.layout.**GridPane**
  - javafx.scene.layout.**HBox**
  - javafx.scene.layout.**StackPane**
  - javafx.scene.layout.**TilePane**
  - javafx.scene.layout.**VBox**

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. You can also specify the gap between the nodes in pixels.

FlowPane lays out nodes row-by-row horizontally or column-by-column vertically. Its data fields **alignment**, **orientation**, **hgap**, and **vgap** are binding properties. Recall that each binding property in JavaFX has a getter method (e.g., getHgap()) that returns its value, a setter method (e.g., setHGap(double)) for setting a value, and a getter method that returns the property itself (e.g., hgapProperty()). For a data field of ObjectProperty type, the value getter method returns a value of type T, and the property getter method returns a property value of type ObjectProperty.

GridPane arranges nodes in a grid (matrix) formation. The nodes are placed in the specified column and row indices. **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. **HBox** lays out its children in a single horizontal row. **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. **AnchorPane** allows the edges of child nodes to be anchored to an offset from the anchor pane's edges. If the anchor pane has a border and/or padding set, the offsets will be measured from the inside edge of those insets.

- **Shapes**

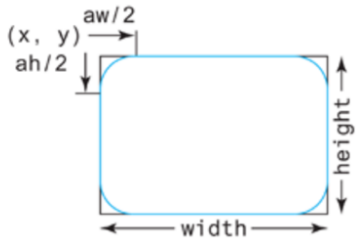
The Shape class is the abstract base class that defines the common properties for all shapes. Among them are the fill, stroke, and strokeWidth properties. The fill property specifies a color that fills the interior of a shape. The stroke property specifies a color that is used to draw the outline of a shape. The strokeWidth property specifies the width of the outline of a shape. The subclasses of Shape are shown below.

- javafx.scene.shape.**Shape**
  - javafx.scene.shape.**Arc**
  - javafx.scene.shape.**Circle**
  - javafx.scene.shape.**CubicCurve**
  - javafx.scene.shape.**Ellipse**
  - javafx.scene.shape.**Line**
  - javafx.scene.shape.**Path**
  - javafx.scene.shape.**Polygon**
  - javafx.scene.shape.**Polyline**
  - javafx.scene.shape.**QuadCurve**
  - javafx.scene.shape.**Rectangle**
  - javafx.scene.shape.**SVGPath**

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.

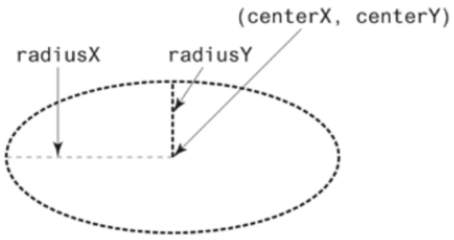
A **line** connects two points with four parameters startX, startY, endX, and endY. The **Line class** defines a line.

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.

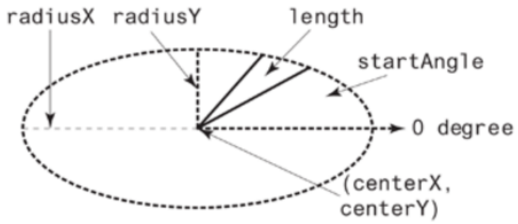


A circle is defined by its parameters centerX, centerY, and radius. The **Circle class** defines a circle.

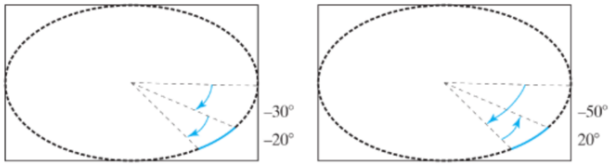
An ellipse is defined by its parameters centerX, centerY, radiusX, and radius. The **Ellipse class** defines an ellipse.



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 (i.e., the angle covered by the arc). Angles are measured in degrees and follow the usual mathematical conventions (i.e., 0 degrees is in the easterly direction and positive angles indicate counterclockwise rotation from the easterly direction).



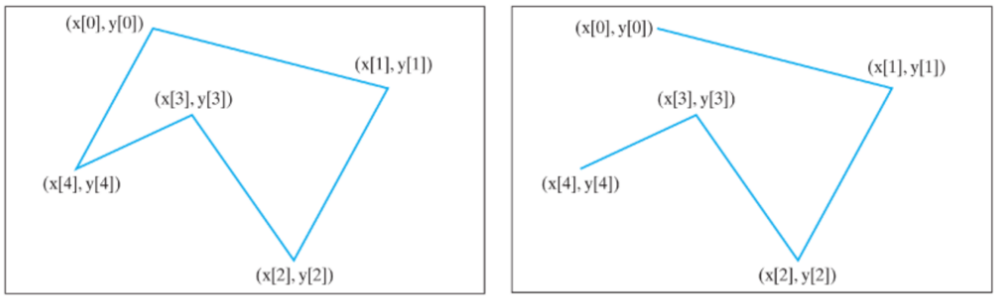
Angles may be negative. A negative starting angle sweeps clockwise from the easterly direction, as shown below. A negative spanning angle sweeps clockwise from the starting angle.



(a) Negative starting angle –30° and negative spanning angle –20°

(b) Negative starting angle –50° and positive spanning angle 20°

The **Polygon class** defines a polygon that connects a sequence of points, and the **Polyline class** is similar to the Polygon class except that the Polyline class is not automatically closed.



(a) Polygon

(b) Polyline