

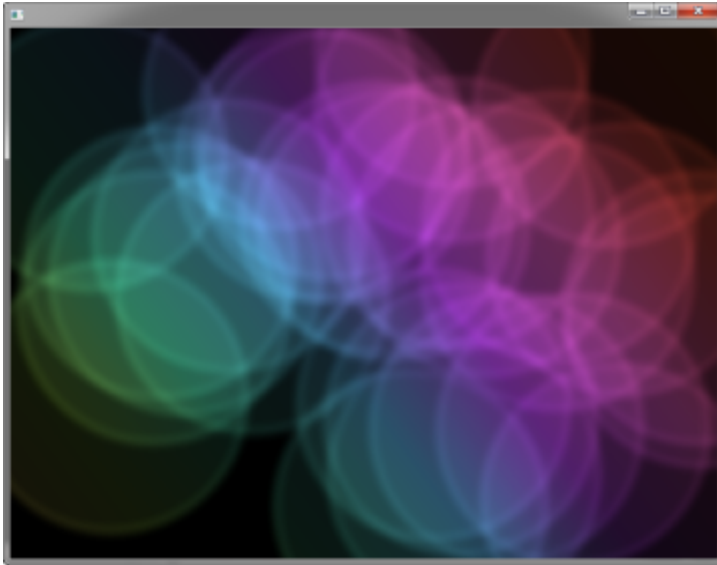
# JavaFX: Getting Started with JavaFX

## 7 Animation and Visual Effects in JavaFX

You can use JavaFX to quickly develop applications with rich user experiences. In this Getting Started tutorial, you will learn to create animated objects and attain complex effects with very little coding.

[Figure 7-1](#) shows the application to be created.

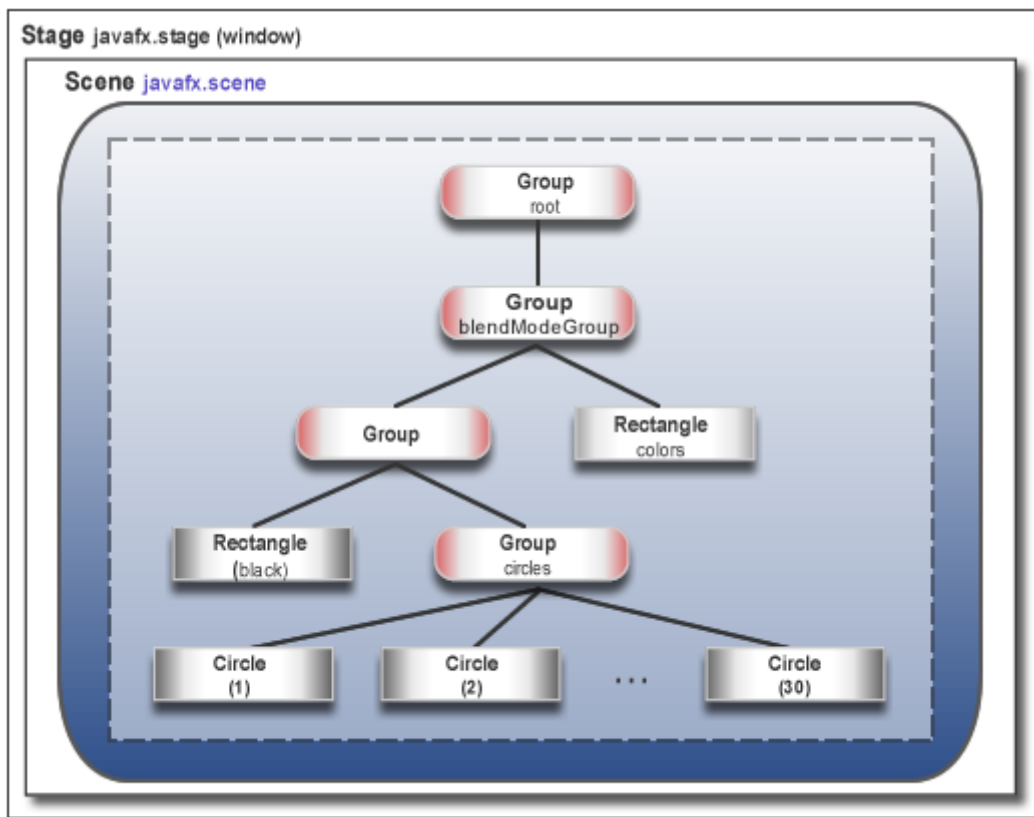
***Figure 7-1 Colorful Circles Application***



Description of "Figure 7-1 Colorful Circles Application"

[Figure 7-2](#) shows the scene graph for the `ColorfulCircles` application. Nodes that branch are instantiations of the `Group` class, and the nonbranching nodes, also known as leaf nodes, are instantiations of the `Rectangle` and `Circle` classes.

***Figure 7-2 Colorful Circles Scene Graph***



[Description of "Figure 7-2 Colorful Circles Scene Graph"](#)

The tool used in this Getting Started tutorial is NetBeans IDE. Before you begin, ensure that the version of NetBeans IDE that you are using supports JavaFX 8. See the Certified System Configurations section of the [Java SE Downloads page](#) for details.

## Set Up the Application

Set up your JavaFX project in NetBeans IDE as follows:

1. From the **File** menu, choose **New Project**.
2. In the **JavaFX** application category, choose **JavaFX Application**. Click **Next**.
3. Name the project **ColorfulCircles** and click **Finish**.
4. Delete the import statements that NetBeans IDE generated.

You can generate the import statements as you work your way through the tutorial by using either the code completion feature or the Fix Imports command from the Source menu in NetBeans IDE. When there is a choice of import statements, choose the one that starts with `javafx`.

## Set Up the Project

Delete the `ColorfulCircles` class from the source file that NetBeans IDE generated and replace it with the code in [Example 7-1](#).

### *Example 7-1 Basic Application*

```

public class ColorfulCircles extends Application {

    @Override
    public void start(Stage primaryStage) {
        Group root = new Group();
        Scene scene = new Scene(root, 800, 600, Color.BLACK);
        primaryStage.setScene(scene);

        primaryStage.show();
    }

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

```

For the `ColorfulCircles` application, it is appropriate to use a group node as the root node for the scene. The size of the group is dictated by the size of the nodes within it. For most applications, however, you want the nodes to track the size of the scene and change when the stage is resized. In that case, you use a resizable layout node as the root, as described in [Creating a Form in JavaFX](#).

You can compile and run the `ColorfulCircles` application now, and at each step of the tutorial, to see the intermediate results. If you run into problems, then take a look at the code in the `ColorfulCircles.java` file, which is included in the downloadable [ColorfulCircles.zip](#) file. At this point, the application is a simple black window.

## Add Graphics

Next, create 30 circles by adding the code in [Example 7-2](#) before the `primaryStage.show()` line.

### *Example 7-2 30 Circles*

```

Group circles = new Group();
for (int i = 0; i < 30; i++) {
    Circle circle = new Circle(150, Color.web("white", 0.05));
    circle.setStrokeType(StrokeType.OUTSIDE);
    circle.setStroke(Color.web("white", 0.16));
    circle.setStrokeWidth(4);
    circles.getChildren().add(circle);
}
root.getChildren().add(circles);

```

This code creates a group named `circles`, then uses a `for` loop to add 30 circles to the group. Each circle has a radius of 150, fill color of `white`, and opacity level of 5 percent, meaning it is mostly transparent.

To create a border around the circles, the code includes the `StrokeType` class. A stroke type of `OUTSIDE` means the boundary of the circle is extended outside the interior by the `StrokeWidth` value, which is 4. The color of the stroke is `white`, and the opacity level is 16 percent, making it brighter than the color of the circles.

The final line adds the `circles` group to the root node. This is a temporary structure. Later, you will modify this scene graph to match the one shown in [Figure 7-2](#).

[Figure 7-3](#) shows the application. Because the code does not yet specify a unique location for each circle, the circles are drawn on top of one another, with the upper left-hand corner of the window as the center point for the circles. The opacity of the overlaid circles interacts with the black background, producing the gray color of the circles.

### *Figure 7-3 Circles*



[Description of "Figure 7-3 Circles"](#)

## Add a Visual Effect

Continue by applying a box blur effect to the circles so that they appear slightly out of focus. The code is in [Example 7-3](#). Add this code before the `primaryStage.show()` line.

### *Example 7-3 Box Blur Effect*

```
circles.setEffect(new BoxBlur(10, 10, 3));
```

This code sets the blur radius to 10 pixels wide by 10 pixels high, and the blur iteration to 3, making it approximate a Gaussian blur. This blurring technique produces a smoothing effect on the edge of the circles, as shown in [Figure 7-4](#).

### *Figure 7-4 Box Blur on Circles*



[Description of "Figure 7-4 Box Blur on Circles"](#)

## Create a Background Gradient

Now, create a rectangle and fill it with a linear gradient, as shown in [Example 7-4](#).

Add the code before the `root.getChildren().add(circles)` line so that the gradient rectangle appears behind the circles.

### *Example 7-4 Linear Gradient*

```
Rectangle colors = new Rectangle(scene.getWidth(), scene.getHeight(),
    new LinearGradient(0f, 1f, 1f, 0f, true, CycleMethod.NO_CYCLE, new
        Stop[]{
            new Stop(0, Color.web("#f8bd55")),
            new Stop(0.14, Color.web("#c0fe56")),
            new Stop(0.28, Color.web("#5dfbc1")),
            new Stop(0.43, Color.web("#64c2f8")),
            new Stop(0.57, Color.web("#be4af7")),
            new Stop(0.71, Color.web("#ed5fc2")),
            new Stop(0.85, Color.web("#ef504c")),
            new Stop(1, Color.web("#f2660f")),
        }));
colors.widthProperty().bind(scene.widthProperty());
colors.heightProperty().bind(scene.heightProperty());
root.getChildren().add(colors);
```

This code creates a rectangle named `colors`. The rectangle is the same width and height as the scene and is filled with a linear gradient that starts in the lower left-hand corner (0, 1) and ends in the upper right-hand corner (1, 0). The value of `true` means the gradient is proportional to the rectangle, and `NO_CYCLE` indicates that the color cycle will not repeat. The `Stop[]` sequence indicates what the gradient color should be at a particular spot.

The next two lines of code make the linear gradient adjust as the size of the scene changes by binding the width and height of the rectangle to the width and height of the scene. See [Using JavaFX Properties and Bindings](#) for more information on binding.

The final line of code adds the `colors` rectangle to the root node.

The gray circles with the blurry edges now appear on top of a rainbow of colors, as shown in [Figure 7-5](#).

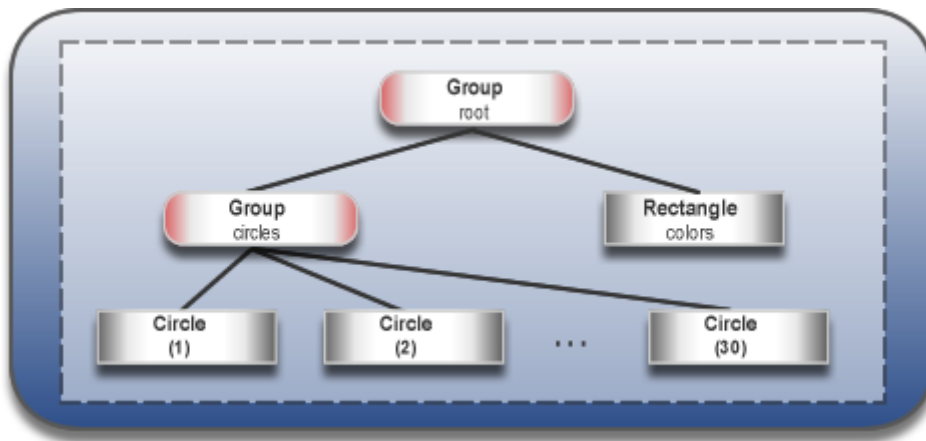
**Figure 7-5 Linear Gradient**



[Description of "Figure 7-5 Linear Gradient"](#)

[Figure 7-6](#) shows the intermediate scene graph. At this point, the `circles` group and `colors` rectangle are children of the root node.

**Figure 7-6 Intermediate Scene Graph**



Description of "Figure 7-6 Intermediate Scene Graph"

## Apply a Blend Mode

Next, add color to the circles and darken the scene by adding an overlay blend effect. You will remove the `circles` group and the linear gradient rectangle from the scene graph and add them to the new overlay blend group.

1. Locate the following two lines of code:

```
root.getChildren().add(colors);
root.getChildren().add(circles);
```

2. Replace the two lines of code from Step 1 with the code in [Example 7-5](#).

### Example 7-5 Blend Mode

```
Group blendModeGroup =
    new Group(new Group(new Rectangle(scene.getWidth(), scene.getHeight(),
        Color.BLACK), circles), colors);
colors.setBlendMode(BlendMode.OVERLAY);
root.getChildren().add(blendModeGroup);
```

The group `blendModeGroup` sets up the structure for the overlay blend. The group contains two children. The first child is a new (unnamed) `Group` containing a new (unnamed) black rectangle and the previously created `circles` group. The second child is the previously created `colors` rectangle.

The `setBlendMode()` method applies the overlay blend to the `colors` rectangle. The final line of code adds the `blendModeGroup` to the scene graph as a child of the root node, as depicted in [Figure 7-2](#).

An overlay blend is a common effect in graphic design applications. Such a blend can darken an image or add highlights or both, depending on the colors in the blend. In this case, the linear gradient rectangle is used as the overlay. The black rectangle serves to keep the background dark, while the nearly transparent circles pick up colors from the gradient, but are also darkened.

[Figure 7-7](#) shows the results. You will see the full effect of the overlay blend when you animate the circles in the next step.

### Figure 7-7 Overlay Blend



[Description of "Figure 7-7 Overlay Blend"](#)

## Add Animation

The final step is to use JavaFX animations to move the circles:

1. If you have not done so already, add `import static java.lang.Math.random;` to the list of import statements.
2. Add the animation code in [Example 7-6](#) before the `primaryStage.show()` line.

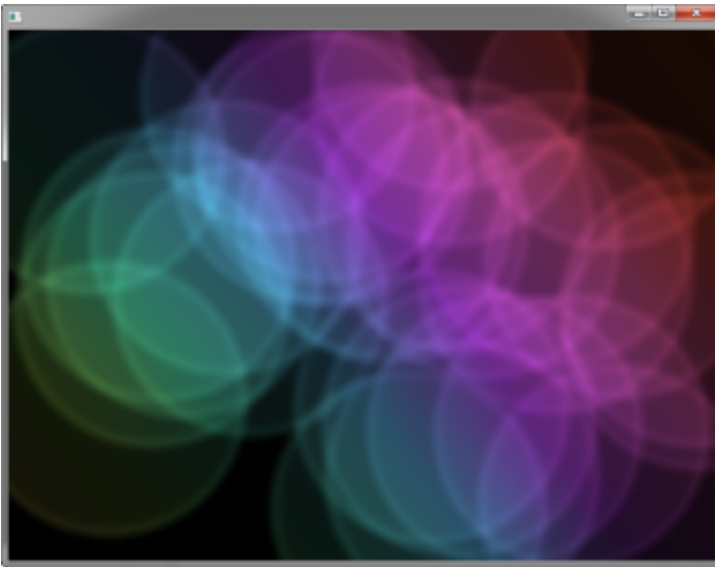
### *Example 7-6 Animation*

```
Timeline timeline = new Timeline();
for (Node circle: circles.getChildren()) {
    timeline.getKeyFrames().addAll(
        new KeyFrame(Duration.ZERO, // set start position at 0
            new KeyValue(circle.translateXProperty(), random() * 800),
            new KeyValue(circle.translateYProperty(), random() * 600)
        ),
        new KeyFrame(new Duration(40000), // set end position at 40s
            new KeyValue(circle.translateXProperty(), random() * 800),
            new KeyValue(circle.translateYProperty(), random() * 600)
        )
    );
}
// play 40s of animation
timeline.play();
```

Animation is driven by a timeline, so this code creates a timeline, then uses a `for` loop to add two key frames to each of the 30 circles. The first key frame at 0 seconds uses the properties `translateXProperty` and `translateYProperty` to set a random position of the circles within the window. The second key frame at 40 seconds does the same. Thus, when the timeline is played, it animates all circles from one random position to another over a period of 40 seconds.

[Figure 7-8](#) shows the 30 colorful circles in motion, which completes the application. For the complete source code, see the `ColorfulCircles.java` file, which is included in the downloadable [ColorfulCircles.zip](#) file..

### *Figure 7-8 Animated Circles*



[Description of "Figure 7-8 Animated Circles"](#)

## Where to Go from Here

Here are several suggestions about what to do next:

- Try the JavaFX samples, which you can download from the JDK Demos and Samples section of the Java SE Downloads page at <http://www.oracle.com/technetwork/java/javase/downloads/>. Especially take a look at the Ensemble application, which provides sample code for animations and effects.
- Read [Creating Transitions and Timeline Animation in JavaFX](#). You will find more details on timeline animation as well as information on fade, path, parallel, and sequential transitions.
- See [Creating Visual Effects in JavaFX](#) for additional ways to enhance the look and design of your application, including reflection, lighting, and shadow effects.
- Try the JavaFX Scene Builder tool to create visually interesting applications. This tool provides a visual layout environment for designing the UI for JavaFX applications and generates FXML code. You can use the Properties panel or the Modify option of the menu bar to add effects to the UI elements. See the Properties Panel and the Menu Bar sections of the JavaFX Scene Builder User Guide for information.