JavaFX

Working with the JavaFX Scene Graph

Release 2.1

E20482-04

September 2013

ORACLE

JavaFX/Working with the JavaFX Scene Graph, Release 2.1

E20482-04

Primary Author: Scott Hommel

Contributing Author:

Contributor:

Contents

1

Working with the JavaFX Scene Graph

JavaFX makes it easy to create modern-looking graphical user interfaces (GUIs) with sophisticated visual effects. This tutorial explores the JavaFX Scene Graph Application Programming Interface (API), the underlying framework that renders your GUI to the screen.

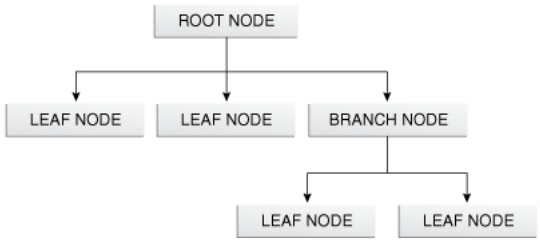
Overview

If you are an experienced Java developer, then chances are good that at some point you have created an application with a graphical user interface. This could be anything from small programs in web pages, to standalone Swing applications on the desktop. And if you have ever done any custom painting, you are familiar with the `Graphics` class and its related methods. The traditional approaches — as powerful as they are — have always required some amount of effort on the developer's part to correctly render the graphics to the screen. This work is often separate from the bulk of the application logic.

The JavaFX Scene Graph API makes graphical user interfaces easier to create, especially when complex visual effects and transformations are involved. A *scene graph* is a tree data structure, most commonly found in graphical applications and libraries such as vector editing tools, 3D libraries, and video games. The JavaFX scene graph is a *retained mode API*, meaning that it maintains an internal model of all graphical objects in your application. At any given time, it knows what objects to display, what areas of the screen need repainting, and how to render it all in the most efficient manner. Instead of invoking primitive drawing methods directly, you instead use the scene graph API and let the system automatically handle the rendering details. This approach significantly reduces the amount of code that is needed in your application.

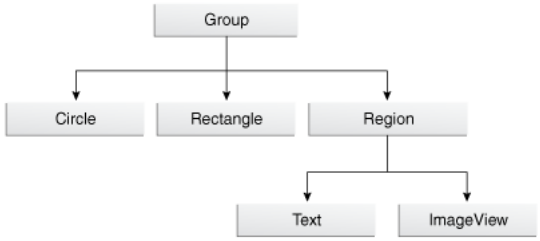
The individual items held within the JavaFX scene graph are known as *nodes*. Each node is classified as either a *branch node* (meaning that it can have children), or a *leaf node* (meaning that it cannot have children). The first node in the tree is always called the *root node*, and it never has a parent. See a general inheritance diagram in Figure 1–1.

Figure 1–1 Root, Branch, and Leaf Nodes



The JavaFX API defines a number of classes that can act as root, branch or leaf nodes. When substituted with actual class names, this same figure might resemble that shown in Figure 1–2 in a real application.

Figure 1–2 Specific Root, Branch, and Leaf Classes



In Figure 1–2, a `Group` object acts as the root node. The `Circle` and `Rectangle` objects are leaf nodes, because they do not (and cannot) have children. The `Region` object (which defines an area of the screen with children than can be styled using CSS) is a branch node that contains two more leaf nodes (`Text` and `ImageView`). Scene graphs can become much larger than this, but the basic organization — that is, the way in which parent nodes contain child nodes — is a pattern that repeats in all applications.

Exploring the API

So what does this all mean in terms of code? Let us start by setting up a basic application frame, populated with only the root node, as shown in Example 1–1.

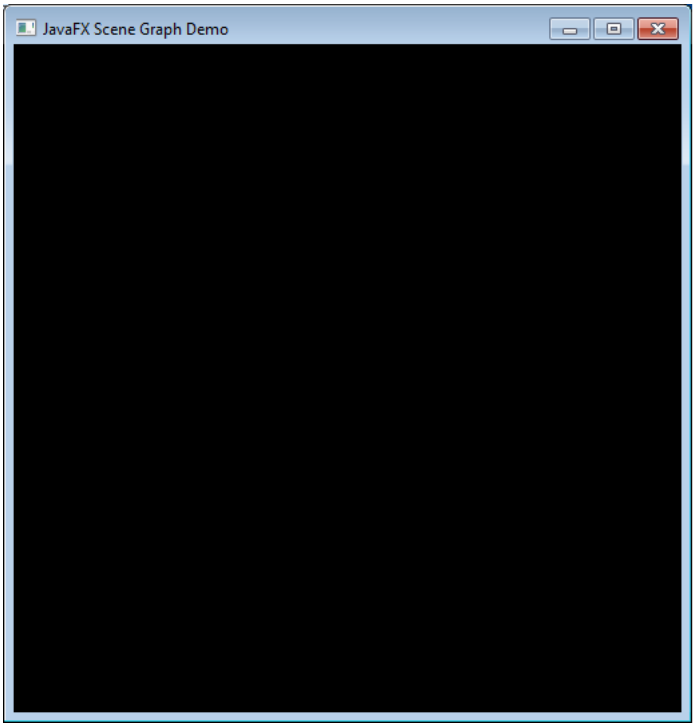
Example 1–1 Creating the Application Frame

```
package scenegraphdemo;

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class Main extends Application {
```

```
    @Override
    public void start(Stage stage) {
        Group root = new Group();
        Scene scene = new Scene(root, 500, 500, Color.BLACK);
        stage.setTitle("JavaFX Scene Graph Demo");
        stage.setScene(scene);
        stage.show();
    }

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

This code produces a window that looks like what is shown in Figure 1–3.

Figure 1–3 Creating a Scene with Root Node Only



The important points to consider are as follows:

1. The `Main` class is an extension of the `javafx.application.Application` class. Its `start` method is overridden and receives a `Stage` object (a top-level GUI container) as its only parameter.
2. The root node (in this case, an instance of the `javafx.scene.Group` class) is created and passed to the scene's constructor, along with the scene's width, height, and fill.
3. The stage's title, scene, and visibility are all set.
4. The main method invokes the `Application.launch()` method.

The resulting application appears as it does because black is the scene's fill color. Because the root node currently has no children, there is nothing else to display.

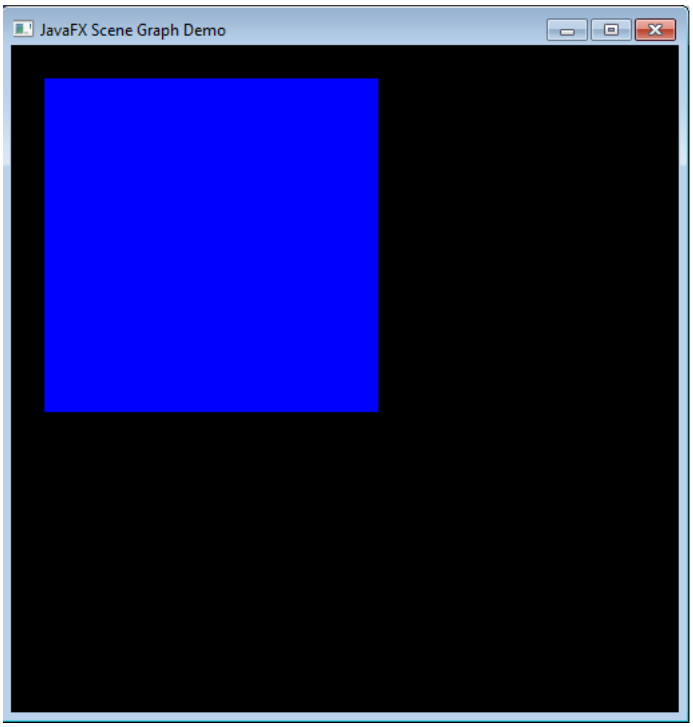
Adding a child to the root node can be accomplished with the modifications shown in Example 1–2.

Example 1–2 Adding a Leaf Node

```
package scenegraphdemo;

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public class Main extends Application {

   @Override
   public void start(Stage stage) {
       Group root = new Group();
       Scene scene = new Scene(root, 500, 500, Color.BLACK);

       Rectangle r = new Rectangle(25,25,250,250);
       r.setFill(Color.BLUE);
       root.getChildren().add(r);

       stage.setTitle("JavaFX Scene Graph Demo");
       stage.setScene(scene);
       stage.show();
   }

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

With the change shown in Example 1–2, a blue rectangle (leaf node) that is 250x250 pixels will appear at the specified X and Y coordinates. (By default, X increases from left to right, and Y increases from top to bottom. This can be affected by transformations, however.) Figure 1–4 shows the result of adding a leaf node.

Figure 1–4 Adding a Leaf Node



Because the graphical objects are managed by the scene graph, you can achieve some interesting effects with very little extra code. For example, you could easily animate the rectangle to bounce back and forth across the screen while rotating, changing its size, and transitioning its color from blue to red.

Example 1–3 uses transitions to make this happen:

Example 1–3 Animating the Scene

```
package scenegraphdemo;

import javafx.animation.FillTransition;
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.animation.Timeline;
import javafx.animation.ParallelTransition;
import javafx.animation.RotateTransition;
import javafx.animation.ScaleTransition;
import javafx.animation.TranslateTransition;
import javafx.util.Duration;

public class Main extends Application {

    @Override
    public void start(Stage stage) {
        Group root = new Group();
        Scene scene = new Scene(root, 500, 500, Color.BLACK);
        Rectangle r = new Rectangle(0, 0, 250, 250);
```

```
        r.setFill(Color.BLUE);
        root.getChildren().add(r);

        TranslateTransition translate =
        new TranslateTransition(Duration.millis(750));
        translate.setToX(390);
        translate.setToY(390);

        FillTransition fill = new FillTransition(Duration.millis(750));
        fill.setToValue(Color.RED);

        RotateTransition rotate = new RotateTransition(Duration.millis(750));
        rotate.setToAngle(360);

        ScaleTransition scale = new ScaleTransition(Duration.millis(750));
        scale.setToX(0.1);
        scale.setToY(0.1);

        ParallelTransition transition = new ParallelTransition(r,
        translate, fill, rotate, scale);
        transition.setCycleCount(Timeline.INDEFINITE);
        transition.setAutoReverse(true);
        transition.play();

        stage.setTitle("JavaFX Scene Graph Demo");
        stage.setScene(scene);
        stage.show();
    }

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

While these examples may be simple, they identify and demonstrate some important concepts that you will use in most graphical applications.

The `javafx.scene` package defines more than a dozen classes, but three in particular are most important when it comes to learning how the API is structured:

- `Node`: The abstract base class for all scene graph nodes.
- `Parent`: The abstract base class for all branch nodes. (This class directly extends `Node`).
- `Scene`: The base container class for all content in the scene graph.

These base classes define important functionality that will subsequently be inherited by subclasses, including paint order, visibility, composition of transformations, support for CSS styling, and so on. You will also find various branch node classes that inherit directly from the `Parent` class, such as `Control`, `Group`, `Region`, and `WebView`. The leaf node classes are defined throughout a number of additional packages, such as `javafx.scene.shape` and `javafx.scene.text`.