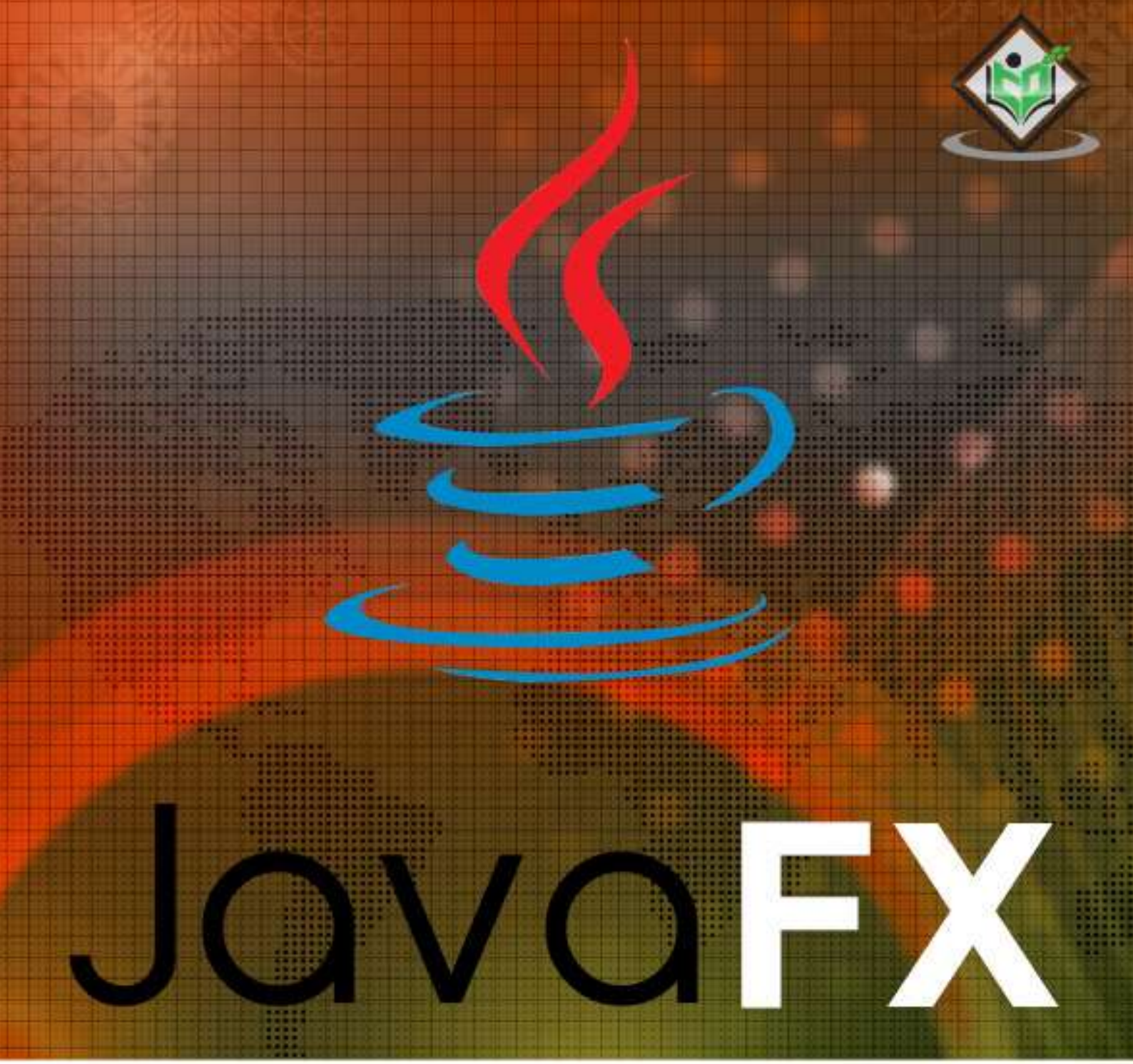


JavaFX

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia https://twitter.com/tutorialspoint

About the Tutorial

JavaFX is a Java library used to build Rich Internet Applications. The applications written using this library can run consistently across multiple platforms. The applications developed using JavaFX can run on various devices such as Desktop Computers, Mobile Phones, TVs, Tablets, etc.

To develop **GUI Applications** using Java programming language, the programmers rely on libraries such as **Advanced Windowing Toolkit** and **Swings**. After the advent of JavaFX, these Java programmers can now develop GUI applications effectively with rich content.

In this tutorial, we will discuss all the necessary elements of JavaFX that you can use to develop effective Rich Internet Applications.

Audience

This tutorial has been prepared for beginners who want to develop Rich Internet Applications using JavaFX.

Prerequisites

For this tutorial, it is assumed that the readers have a prior knowledge of Java programming language.

Copyright & Disclaimer

Table of Contents

1. JavaFX – Overview

Rich Internet Applications are those web applications which provide similar features and experience as that of desktop applications. They offer a better visual experience when compared to the normal web applications to the users. These applications are delivered as browser plug-ins or as a virtual machine and are used to transform traditional static applications into more enhanced, fluid, animated and engaging applications.

Unlike traditional desktop applications, RIA's don't require to have any additional software to run. As an alternative, you should install software such as ActiveX, Java, Flash, depending on the Application.

In an RIA, the graphical presentation is handled on the client side, as it has a plugin that provides support for rich graphics. In a nutshell, data manipulation in an RIA is carried out on the server side, while related object manipulation is carried out on the client side.

We have three main technologies using which we can develop an RIA. These include the following:

- Adobe Flash
- Microsoft Silverlight
- JavaFX

Adobe Flash

This software platform is developed by Adobe Systems and is used in creating Rich Internet Applications. Along with these, you can also build other Applications such as Vector, Animation, Browser Games, Desktop Applications, Mobile Applications and Games, etc.

This is the most commonly used platform for developing and executing RIA's with a desktop browser penetration rate of 96%.

Microsoft Silverlight

Just like Adobe flash, Microsoft Silverlight is also a software application framework for developing as well as executing Rich Internet Applications. Initially this framework was used for streaming media. The present versions support multimedia, graphics, and animation as well.

This platform is rarely used with a desktop browser penetration rate of 66%.

JavaFX

JavaFX is a Java library using which you can develop Rich Internet Applications. By using Java technology, these applications have a browser penetration rate of 76%.

What is JavaFX?

JavaFX is a Java library used to build Rich Internet Applications. The applications written using this library can run consistently across multiple platforms. The applications developed using JavaFX can run on various devices such as Desktop Computers, Mobile Phones, TVs, Tablets, etc.

To develop **GUI Applications** using Java programming language, the programmers rely on libraries such as **Advanced Windowing Toolkit** and **Swings**. After the advent of JavaFX, these Java programmers can now develop GUI applications effectively with rich content.

Need for JavaFX

To develop **Client Side Applications** with rich features, the programmers used to depend on various libraries to add features such as Media, UI controls, Web, 2D and 3D, etc. JavaFX includes all these features in a single library. In addition to these, the developers can also access the existing features of a Java library such as **Swings**.

JavaFX provides a rich set of graphics and media API's and it leverages the modern **Graphical Processing Unit** through hardware accelerated graphics. JavaFX also provides interfaces using which developers can combine graphics animation and UI control.

One can use JavaFX with JVM based technologies such as Java, Groovy and JRuby. If developers opt for JavaFX, there is no need to learn additional technologies, as prior knowledge of any of the above-mentioned technologies will be good enough to develop RIA's using JavaFX.

Features of JavaFX

Following are some of the important features of JavaFX –

- **Written in Java:** The JavaFX library is written in Java and is available for the languages that can be executed on a JVM, which include – **Java, Groovy** and **JRuby**. These JavaFX applications are also platform independent.

- **FXML**: JavaFX features a language known as FXML, which is a HTML like declarative markup language. The sole purpose of this language is to define a user Interface.

- **Scene Builder**: JavaFX provides an application named Scene Builder. On integrating this application in IDE's such as Eclipse and NetBeans, the users can access a drag and drop design interface, which is used to develop FXML applications (just like Swing Drag & Drop and DreamWeaver Applications).

- **Swing Interoperability:** In a JavaFX application, you can embed Swing content using the **Swing Node** class. Similarly, you can update the existing Swing applications with JavaFX features like embedded web content and rich graphics media.

- **Built-in UI controls:** JavaFX library caters UI controls using which we can develop a full-featured application.

- **CSS like Styling:** JavaFX provides a CSS like styling. By using this, you can improve the design of your application with a simple knowledge of CSS.

- **Canvas and Printing API:** JavaFX provides Canvas, an immediate mode style of rendering API. Within the package **javafx.scene.canvas** it holds a set of classes for canvas, using which we can draw directly within an area of the JavaFX scene. JavaFX also provides classes for Printing purposes in the package **javafx.print.**

- **Rich set of API's:** JavaFX library provides a rich set of API's to develop GUI applications, 2D and 3D graphics, etc. This set of API's also includes capabilities of Java platform. Therefore, using this API, you can access the features of Java languages such as Generics, Annotations, Multithreading, and Lambda Expressions. The traditional Java Collections library was enhanced and concepts like observable lists and maps were included in it. Using these, the users can observe the changes in the data models.

- **Integrated Graphics library:** JavaFX provides classes for **2d** and **3d** graphics.

- **Graphics pipeline:** JavaFX supports graphics based on the Hardware-accelerated graphics pipeline known as Prism. When used with a supported Graphic Card or GPU it offers smooth graphics. In case the system does not support graphic card then prism defaults to the software rendering stack.

History of JavaFX

JavaFX was originally developed by **Chris Oliver**, when he was working for a company named **See Beyond Technology Corporation,** which was later acquired by **Sun Microsystems** in the year 2005.

The following points give us more information of this project:

- Initially this project was named as F3 (**Form Follows Functions**) and it was developed with an intention to provide richer interfaces for developing GUI Applications.

- **Sun Microsystems** acquired the See Beyond company in June 2005, it adapted the F3 project as **JavaFX.**

- In the year 2007, JavaFX was announced officially at **Java One**, a world wide web conference which is held yearly.

- In the year 2008, **Net Beans** integrated with JavaFX was available. In the same year, the Java **Standard Development Kit** for JavaFX 1.0 was released.

- In the year 2009, Oracle Corporation acquired Sun Microsystems and in the same year the next version of JavaFX (1.2) was released as well.

- In the year 2010, JavaFX 1.3 came out and in the year 2011 JavaFX 2.0 was released.

- The latest version, JavaFX8, was released as an integral part of Java on 18th of March 2014.

2. JavaFX – Environment

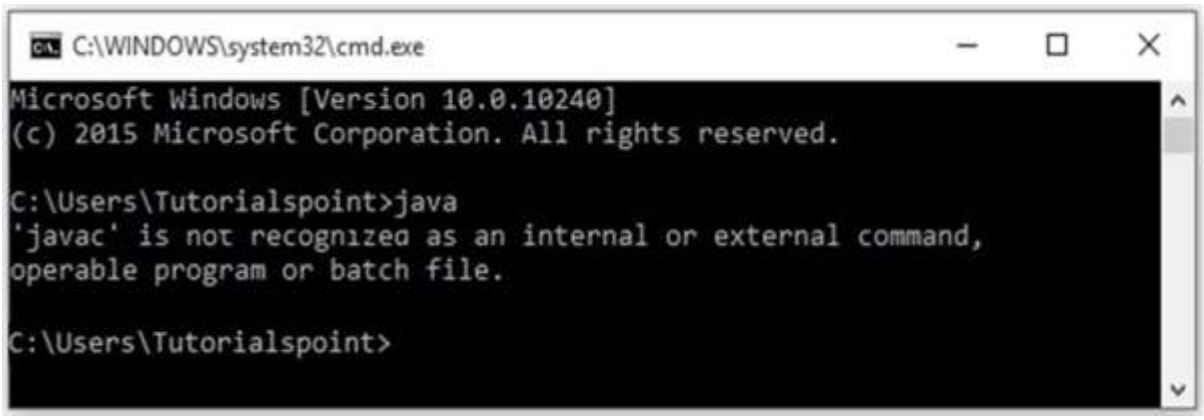
From Java8 onwards, the JDK **(Java Development Kit)** includes **JavaFX** library in it. Therefore, to run JavaFX applications, you simply need to install Java8 or later version in your system.

In addition to it, IDE's like Eclipse and NetBeans provide support for JavaFX. This chapter teaches you how to set the environment to run JavaFX Applications in various ways.

Installing Java8

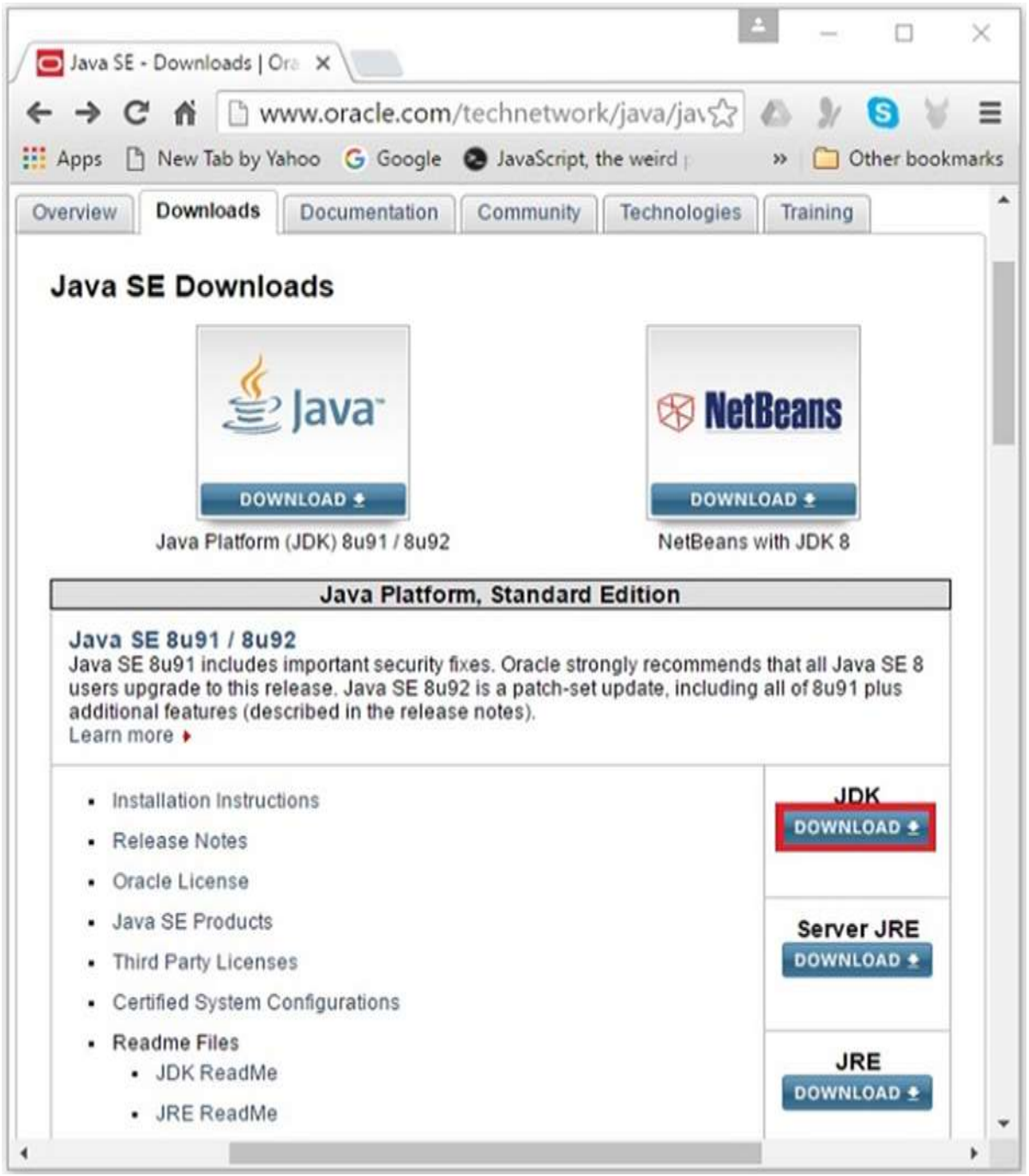
First of all, you will have to verify whether there is Java Installed in your system or not by opening the command prompt and typing the command "Java" in it.

If you haven't installed Java in your system, the command prompt displays the message shown in the following screenshot.



Then install Java by following the steps given below.

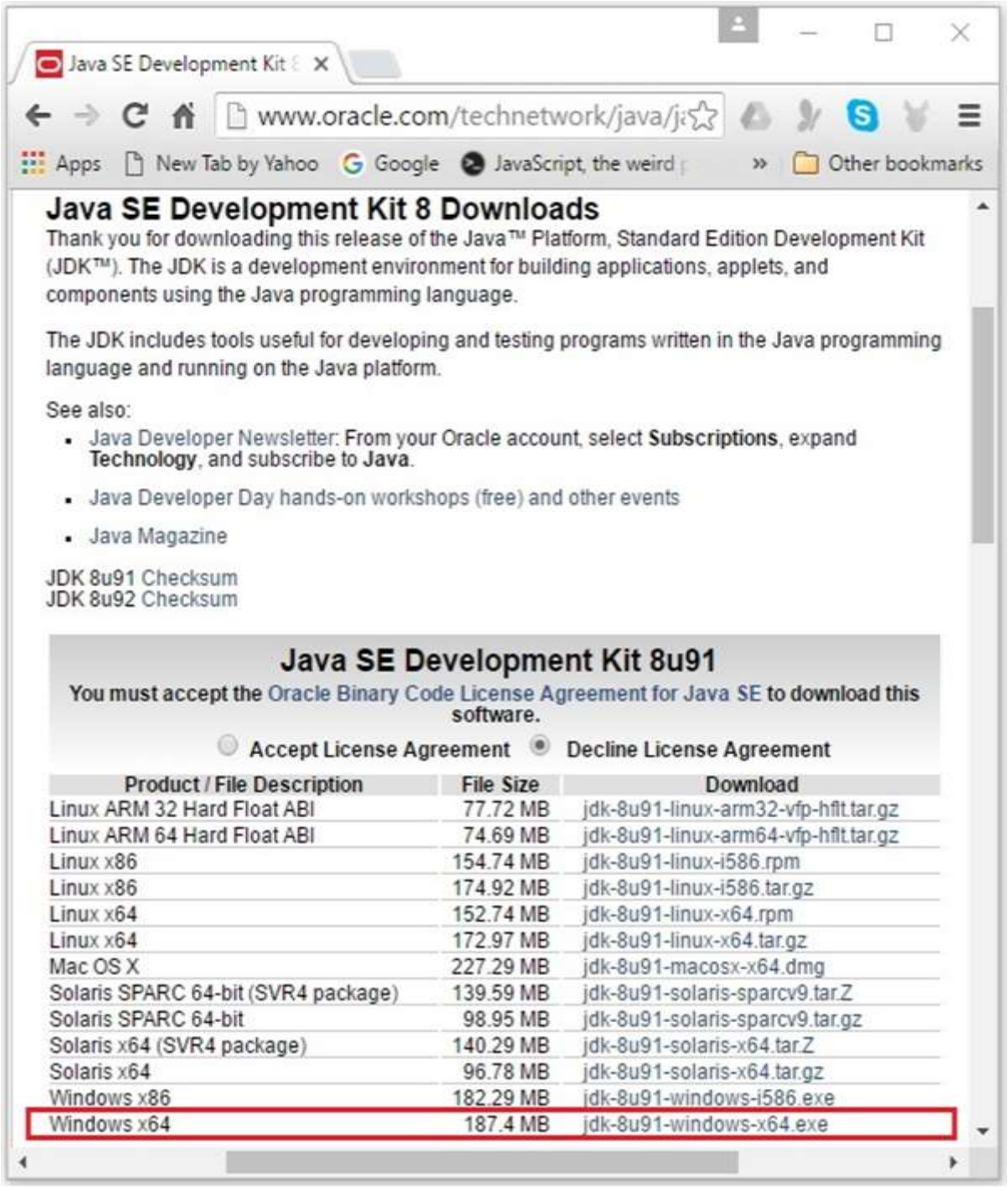
Step 1: Visit the JavaSE Downloads Page, click on the JDK **Download** button as highlighted in the following screenshot.



Step 2: On clicking the Download button, you will be redirected to the **Java SE Development Kit 8 Downloads** page. This page provides you links of JDK for various platforms.

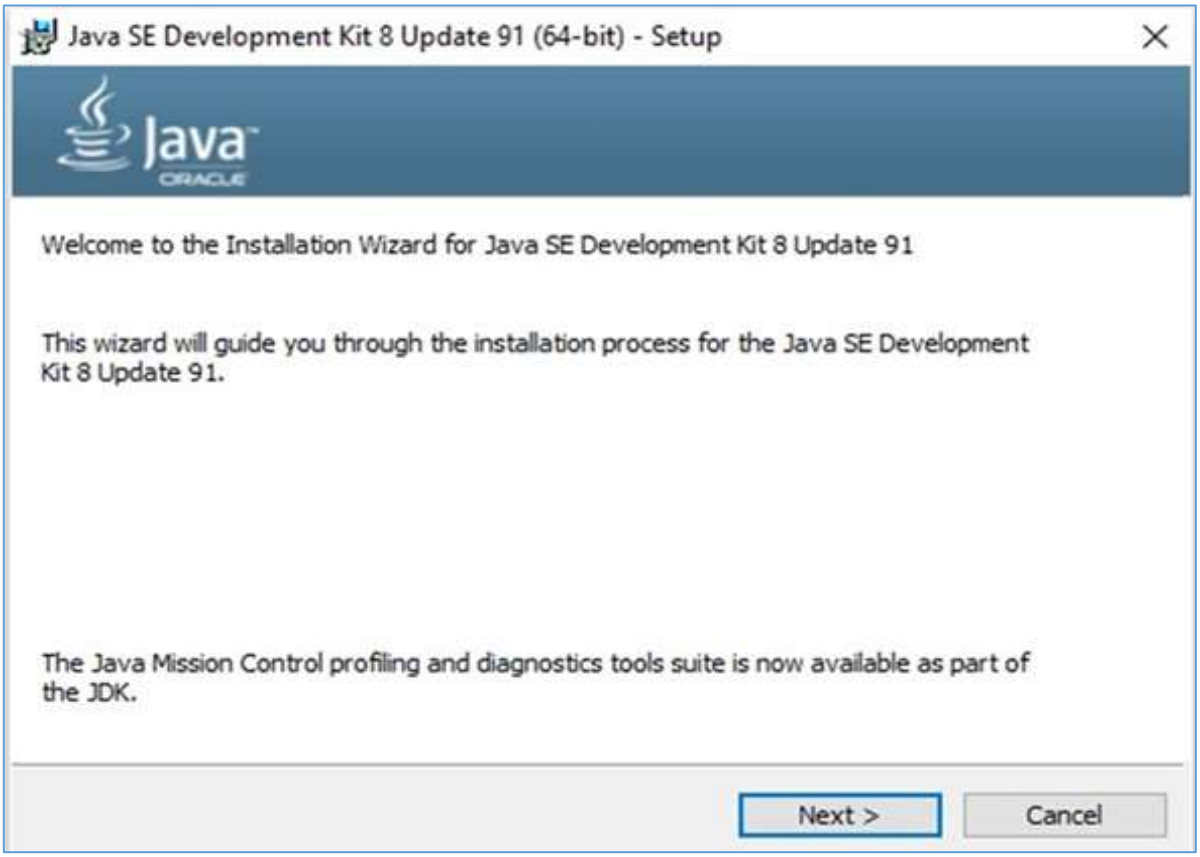
Accept the license agreement and download the required software by clicking on its respective link.

For example, if you are working on a windows 64-bit Operating System then you need to download the JDK version highlighted in the following screenshot.

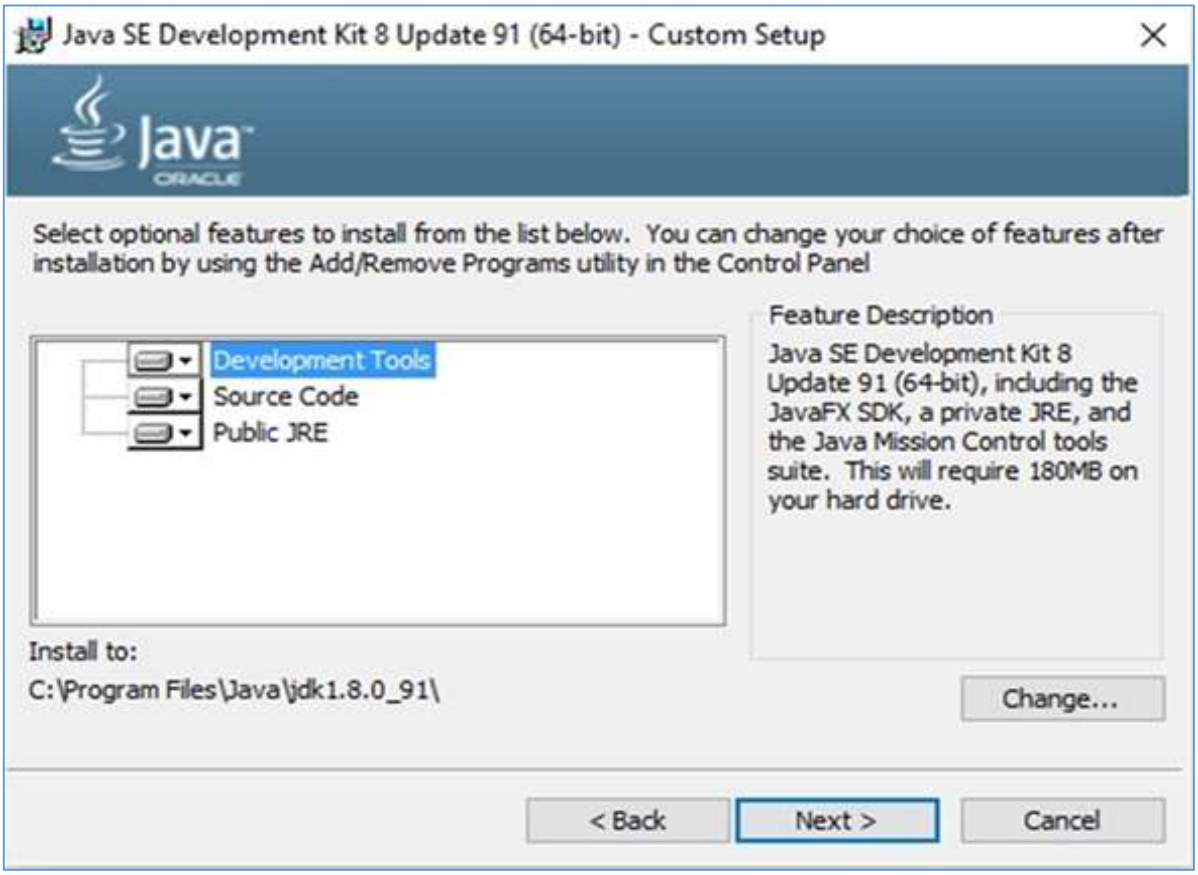


On clicking the highlighted link, the Java8 Development Kit suitable for Windows 64-bit Operating System will be downloaded onto your system.

Step 3: Run the downloaded binary executable file to start the installation of JDK8.

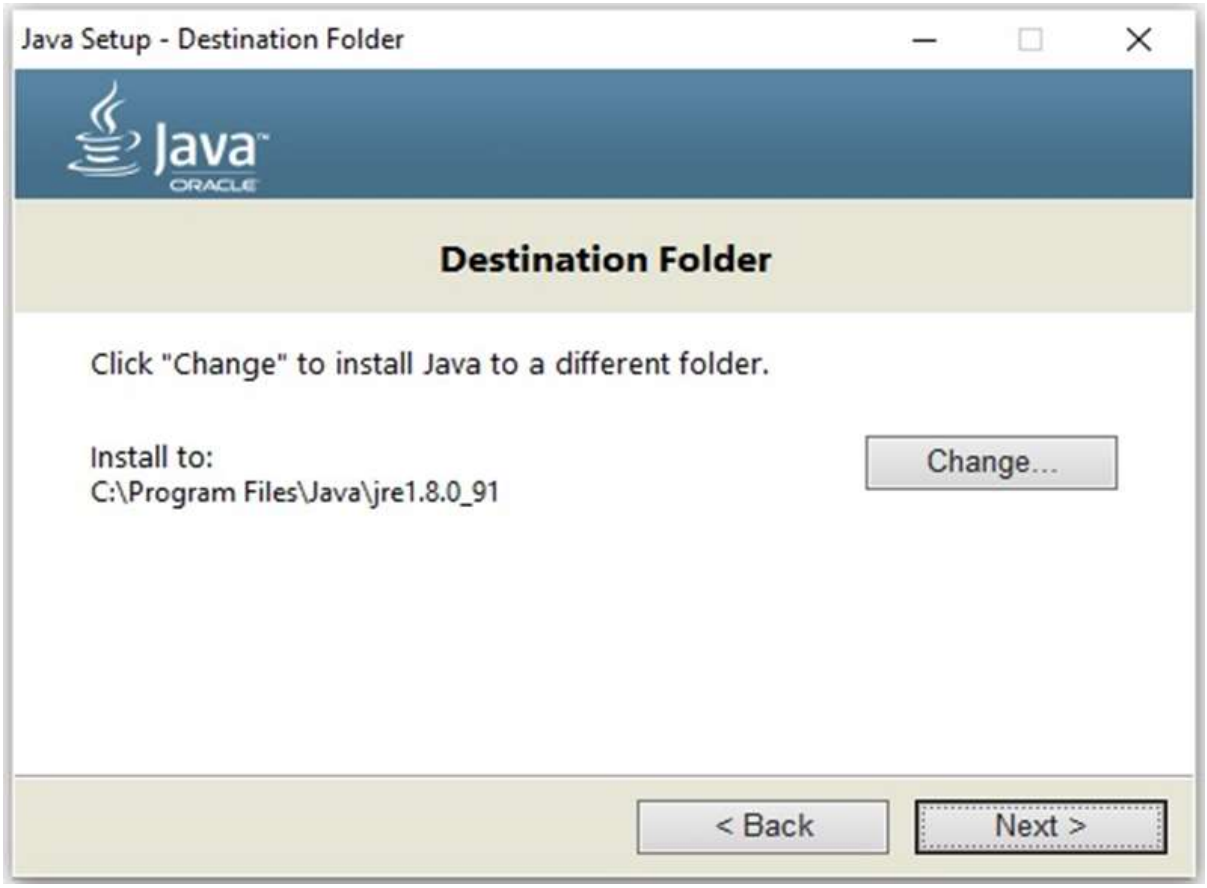


Step 4: Choose the Installation Directory.

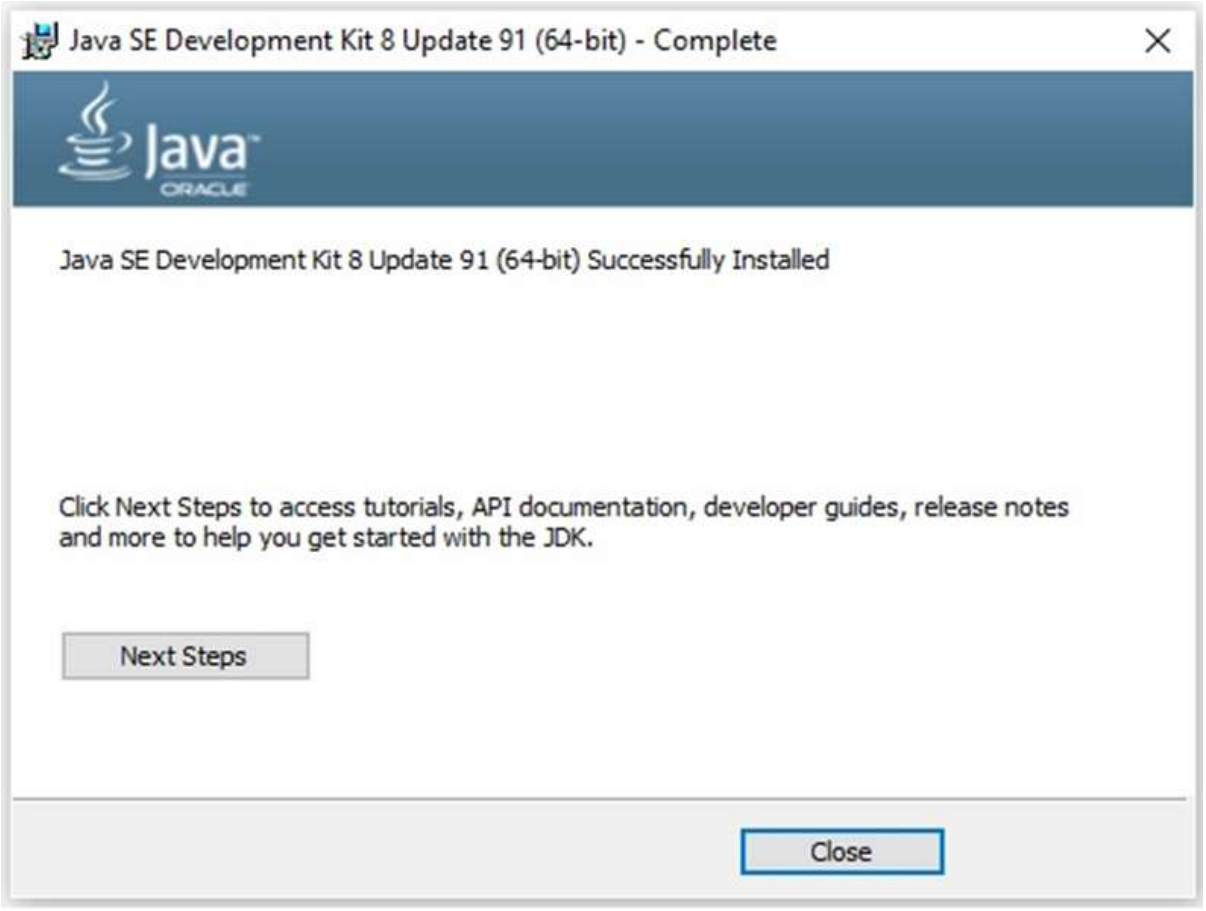


Step 5: On selecting the destination folder and clicking Next, the JavaFX installation process starts displaying the progress bar as shown in the following screenshot.

Step 6: Change the installation directory if needed, else keep the default ones and proceed further.



Step 7: Finish the installation process by clicking the Close button as shown in the following screenshot**.**

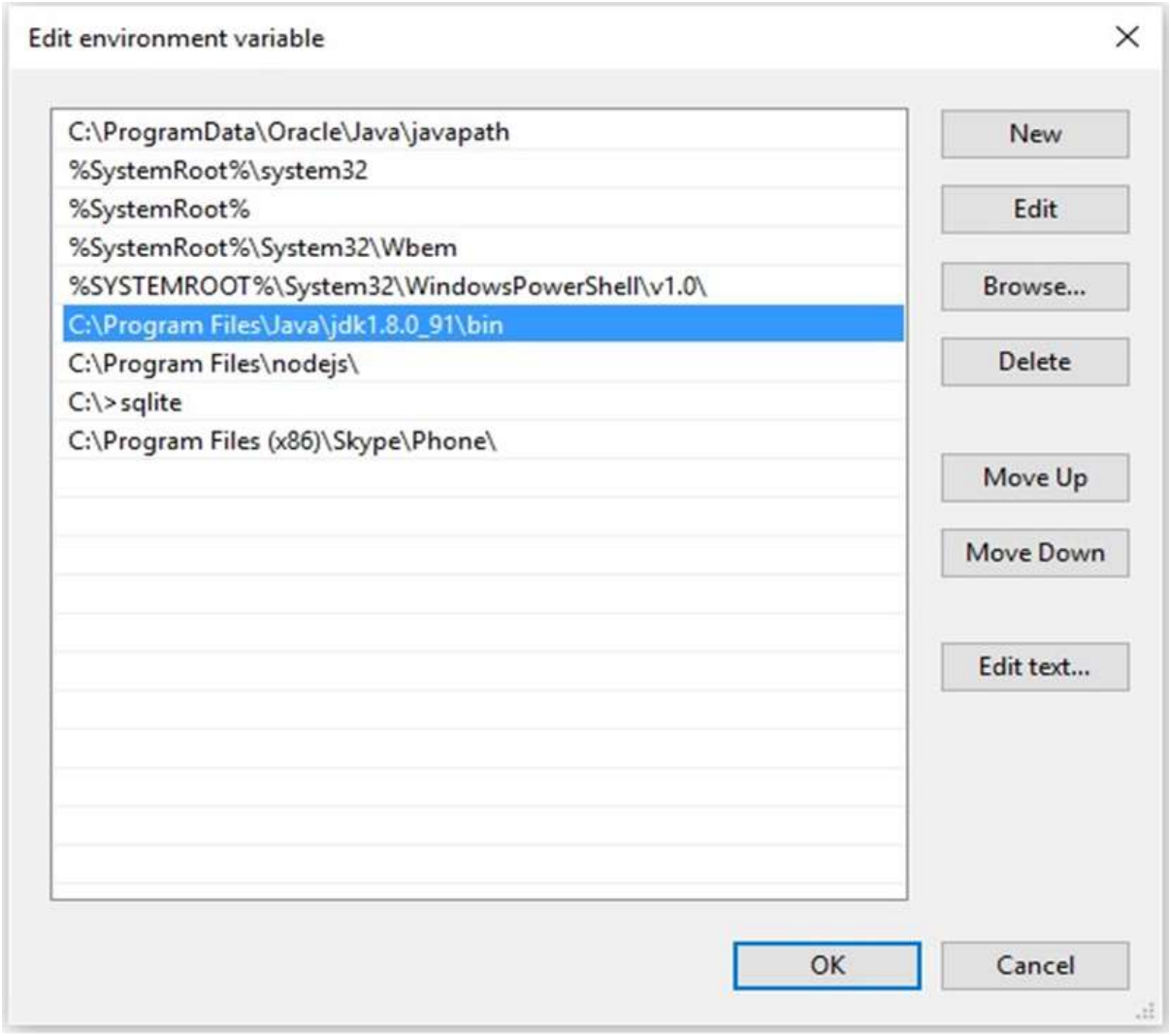


Setting up the Path for Windows

After installing Java, you need to set the path variables. Assume that you have installed Java in **C:\Program Files\java\jdk1.8.0_91** directory.

Now you can follow the steps that are given below.

- Right-click on 'My Computer' and select 'Properties'.
- Click on the 'Environment Variables' button under the 'Advanced' tab.
- Now, alter the 'Path' variable so that it also contains the path to the Java executable. For Example, if the path is currently set to 'C:\WINDOWS\SYSTEM32', then change your path to read 'C:\WINDOWS\SYSTEM32; C:\Program Files\java\ jdk1.8.0_91\bin'.

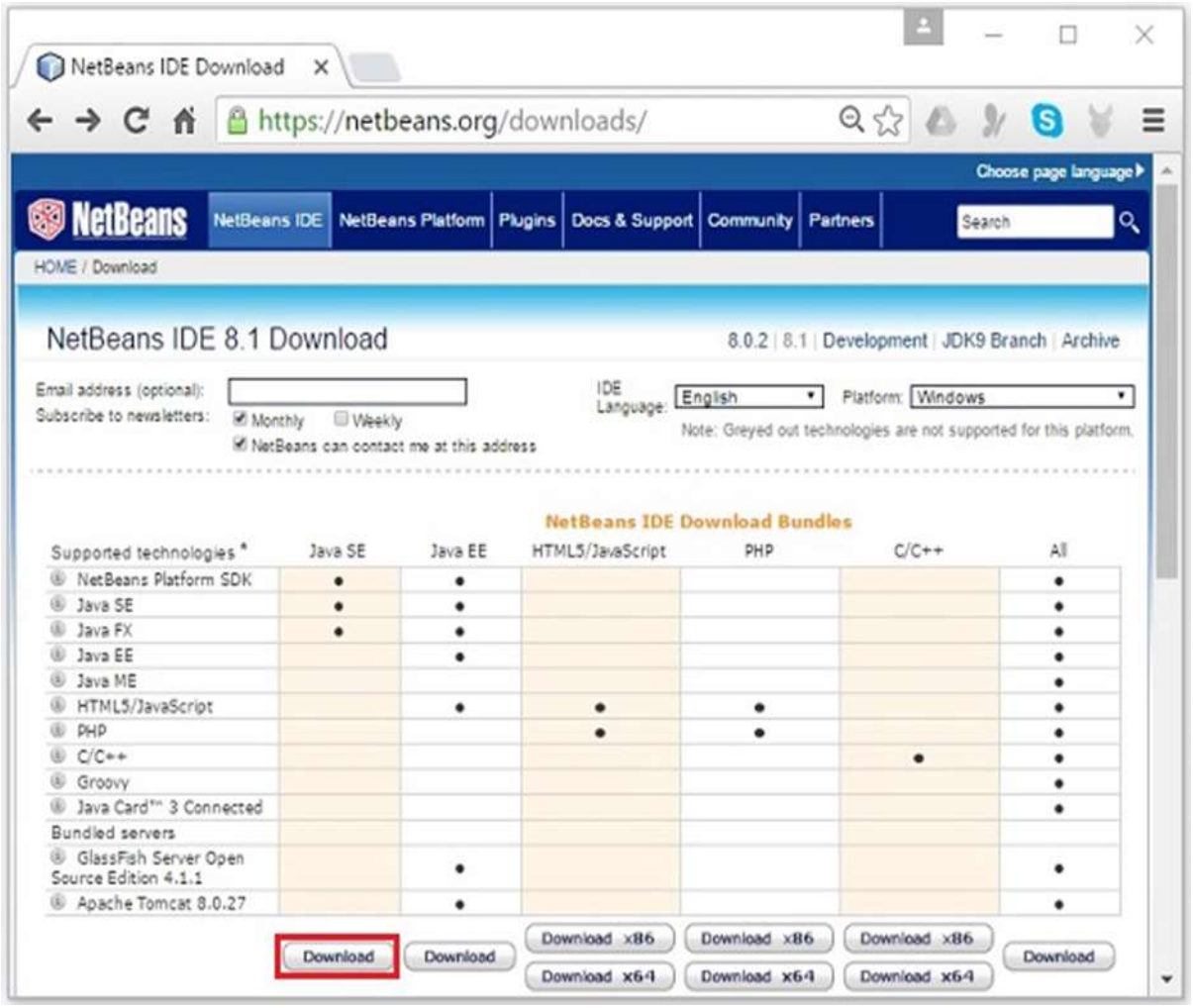


Setting NetBeans Environment of JavaFX

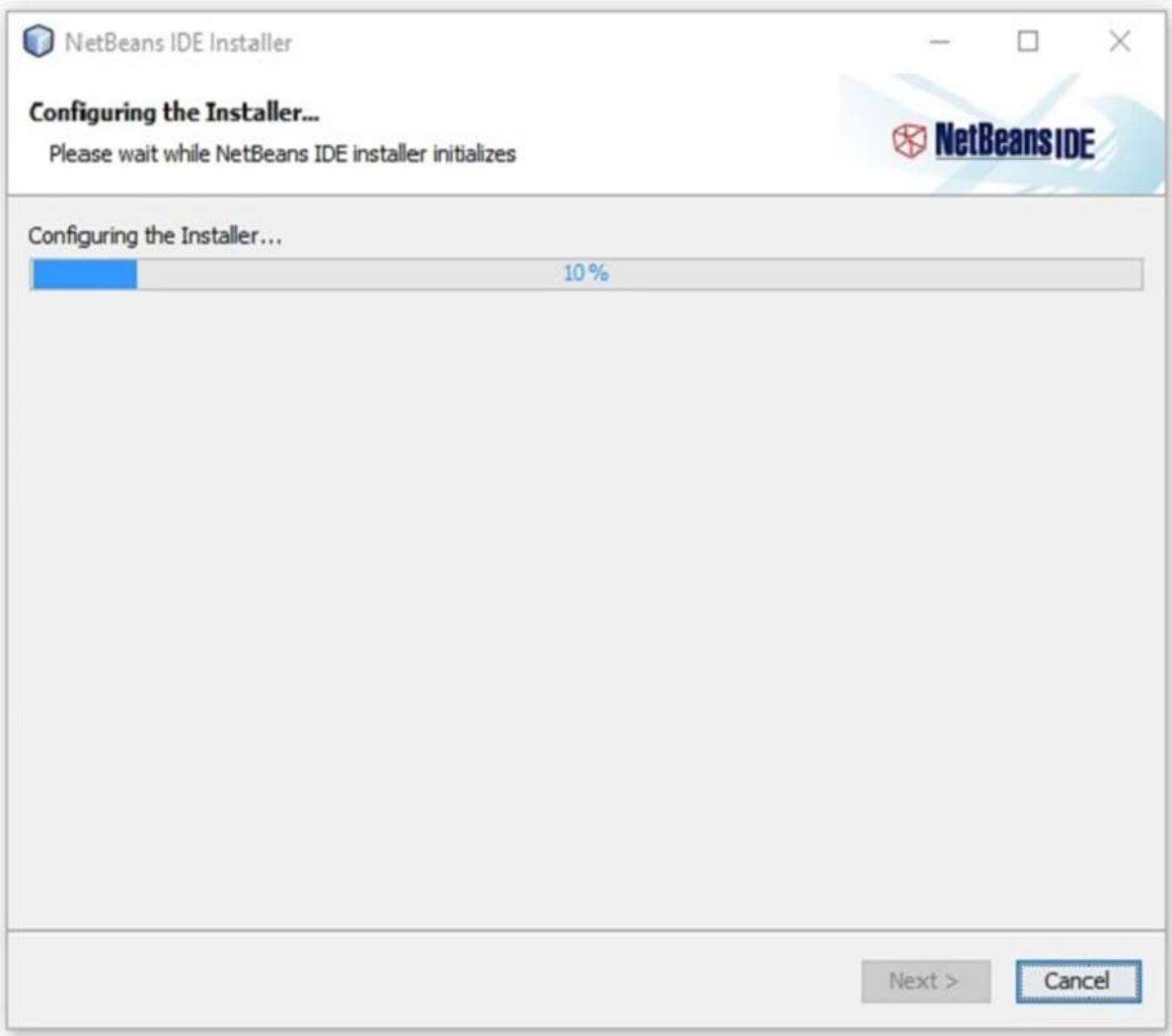
NetBeans8 provides inbuilt support for JavaFX. On installing this, you can create a JavaFX application without any additional plugins or JAR files. To set up the NetBeans environment, you will need to follow the steps that are given below.

Step 1: Visit the NetBeans website and click the Download button in order to download the NetBeans software.

Step 2: On clicking **Download,** you will get to the Downloads page of the NetBeans software, which provides NetBeans bundles for various Java applications. Download the NetBeans software for **JavaSE** as shown in the following screenshot.

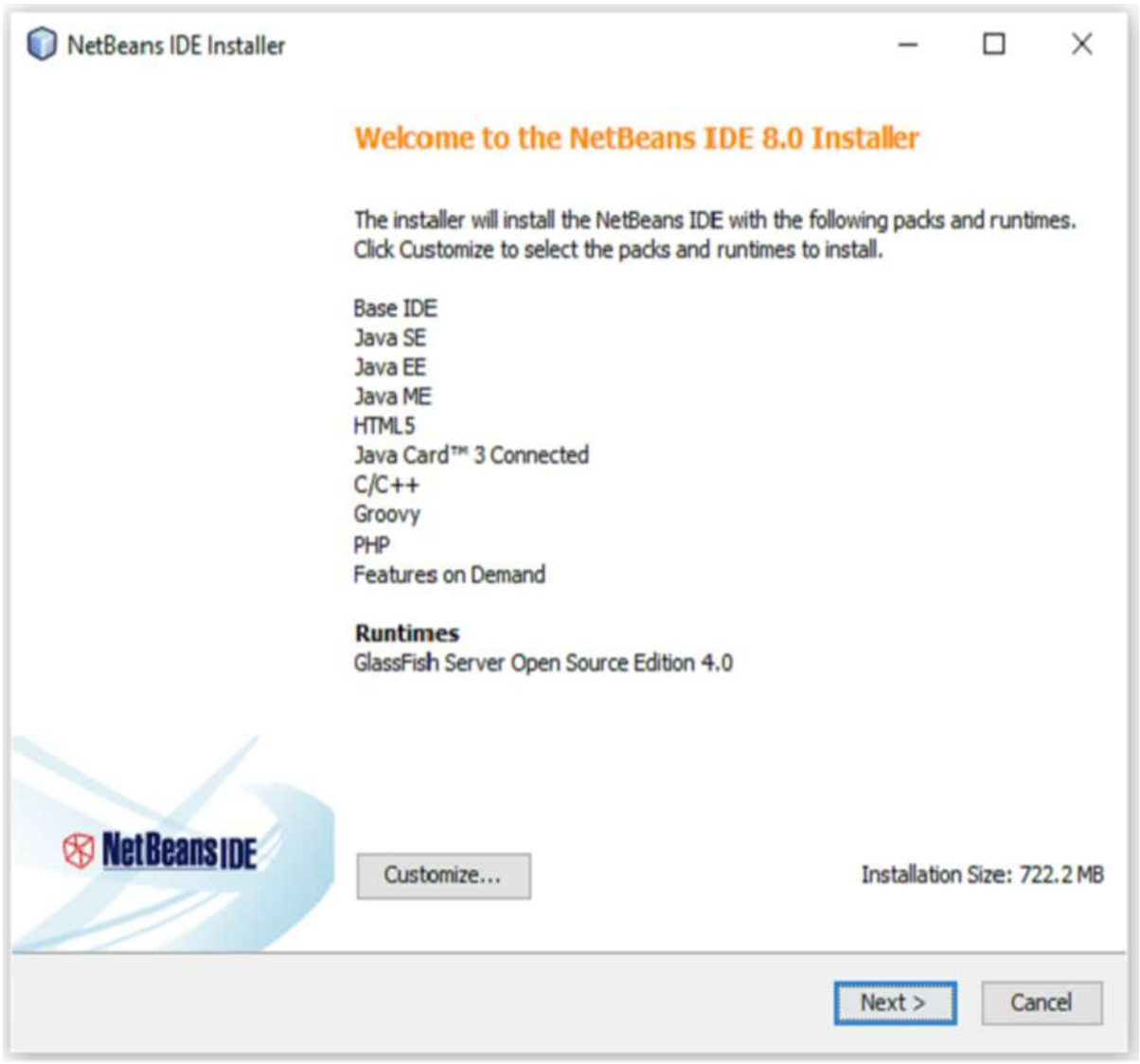


Step 3: On clicking this button, a file named **netbeans-8.0-windows.exe** will be downloaded onto your system. Run this file in order to install it. On running this file, a NetBeans installer will start as shown in the following screenshot.

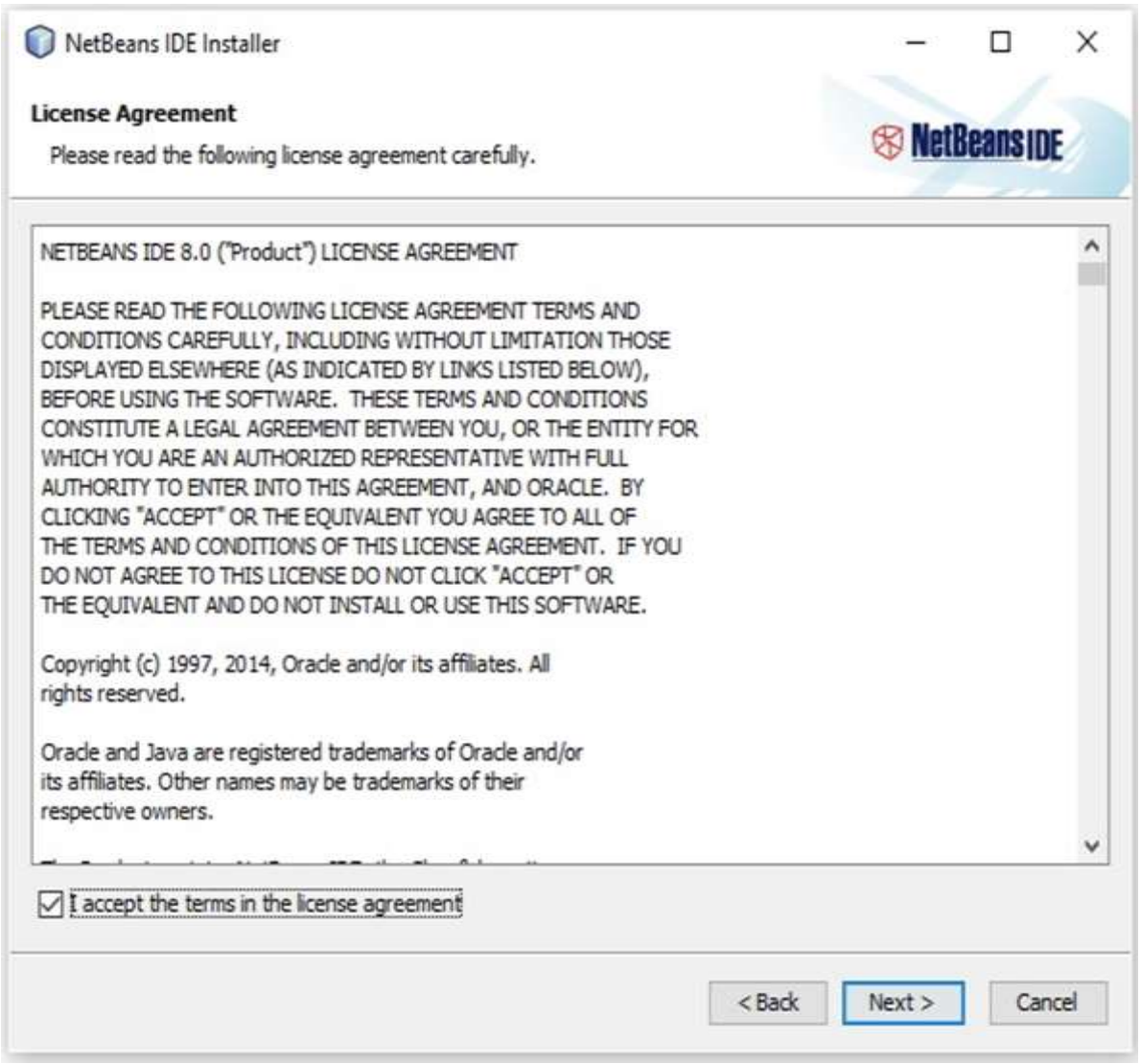


After completion of the configuration, you will see the **Welcome Page** of the installer.

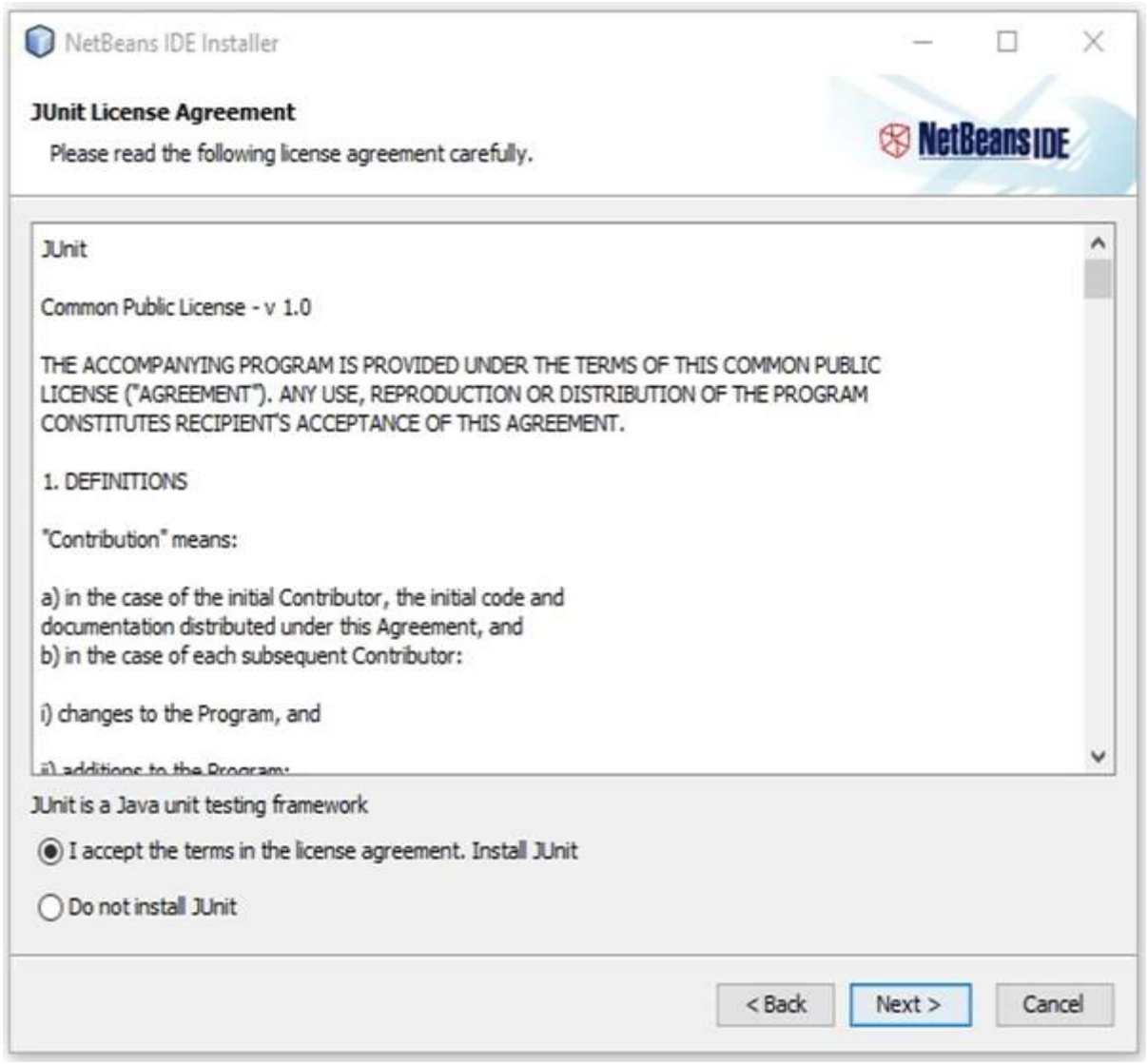
Step 4: Click the Next button and proceed with the installation.



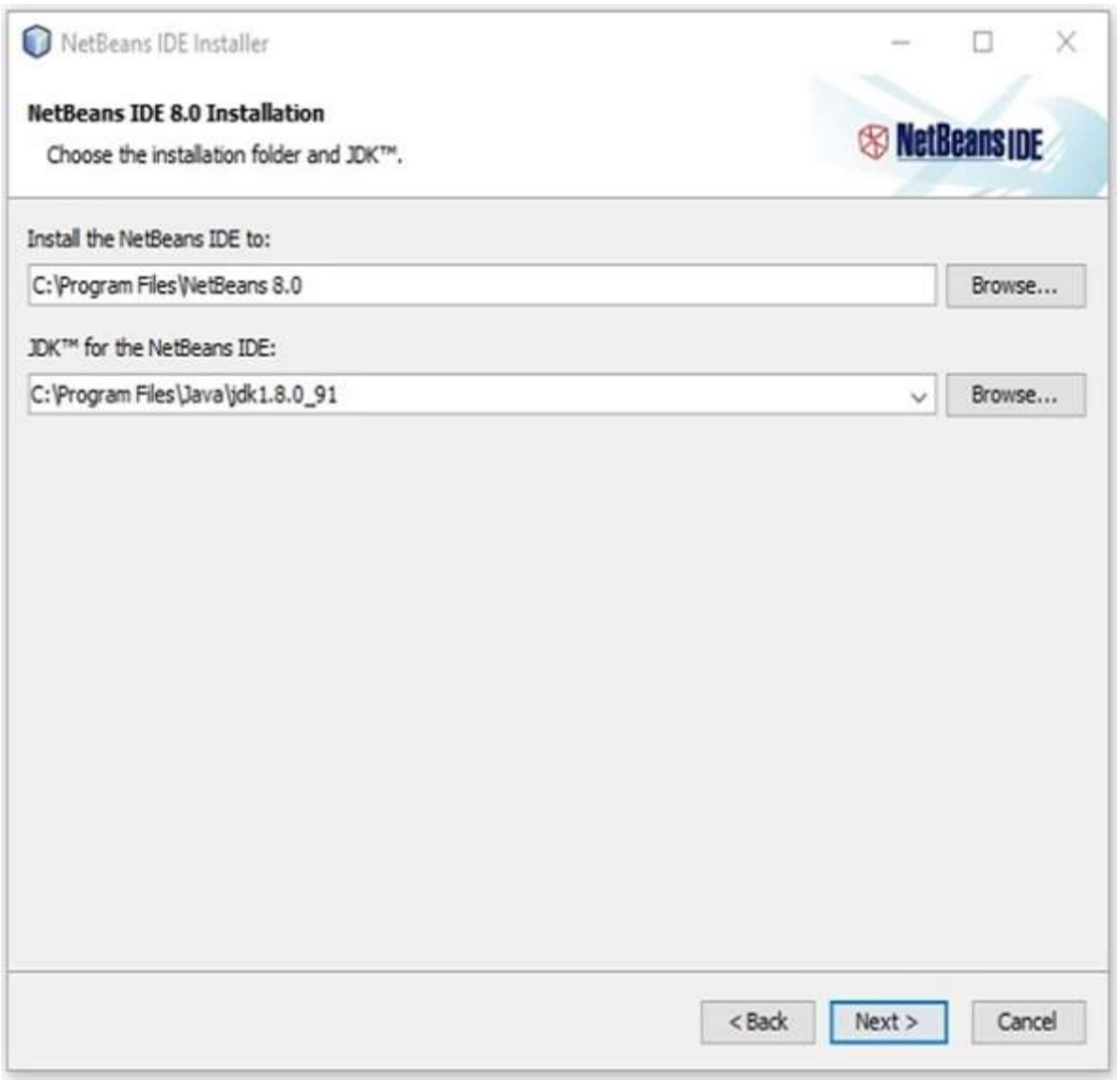
Step 5: The next window holds the **NETBEANS IDE 8.0 license agreement**. Read it carefully and accept the agreement by checking the checkbox at "I accept the terms in the license agreement" and then click the **Next** button.



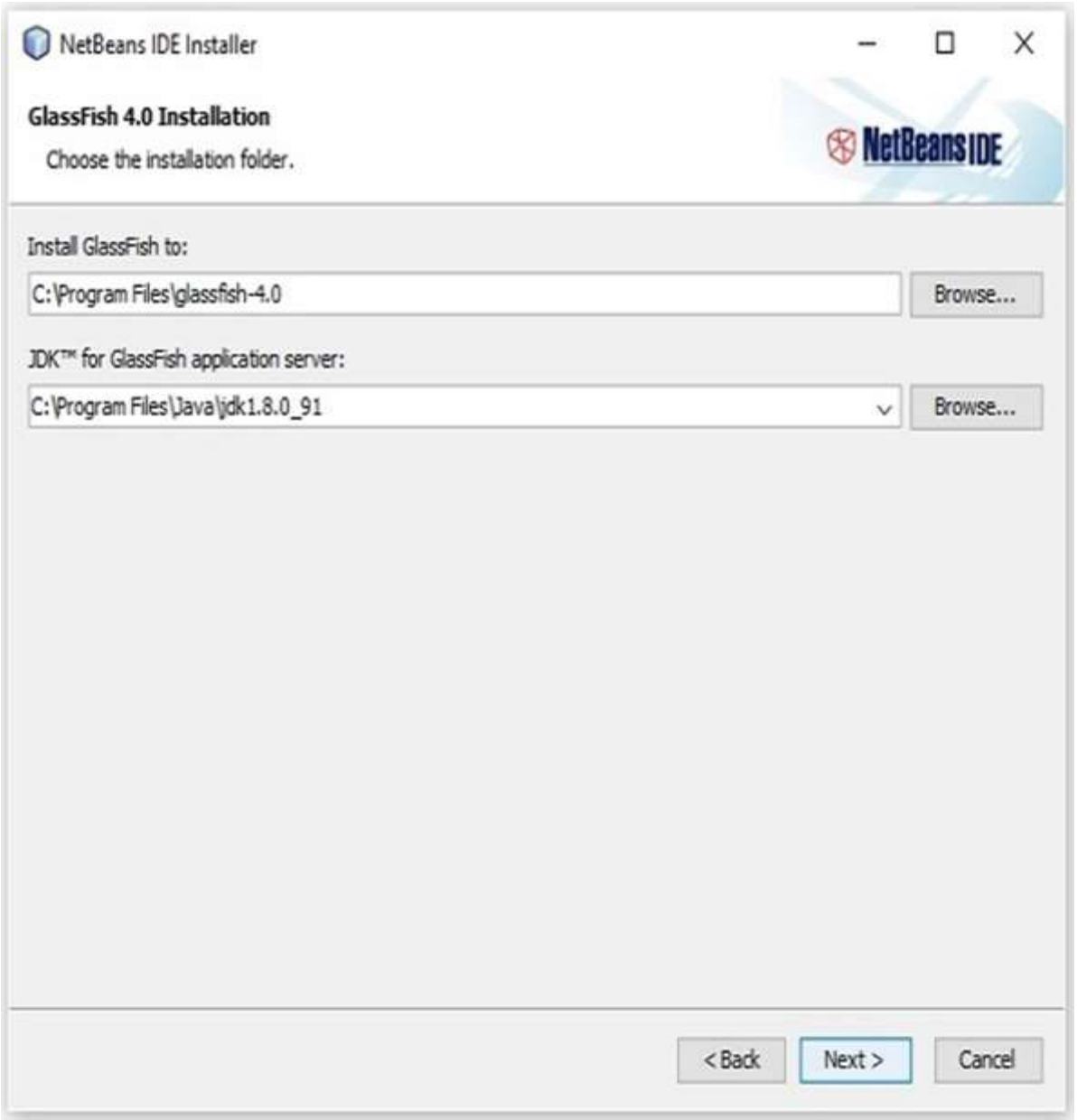
Step 6: In the next window, you will encounter the license agreement for **Junit,** accept it by selecting the radio button at **"**I accept the terms in the license agreement, Install JUnit**" and** click on **Next.**



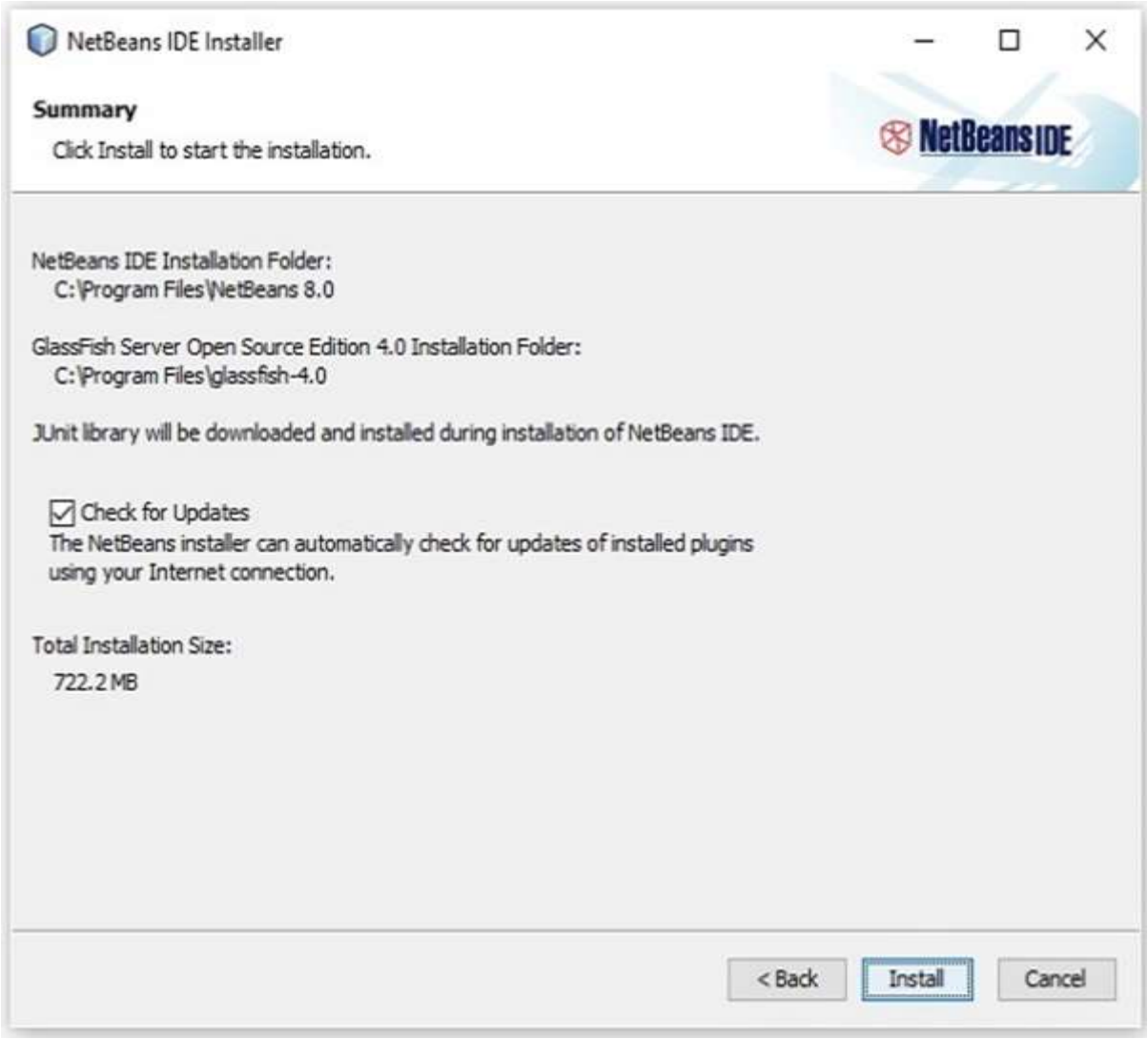
Step 7: Choose the destination directory where you need the Netbeans 8.0 to be installed. Furthermore, you can also browse through the directory where **Java Development Kit** is installed in your system and click on the **Next** button.



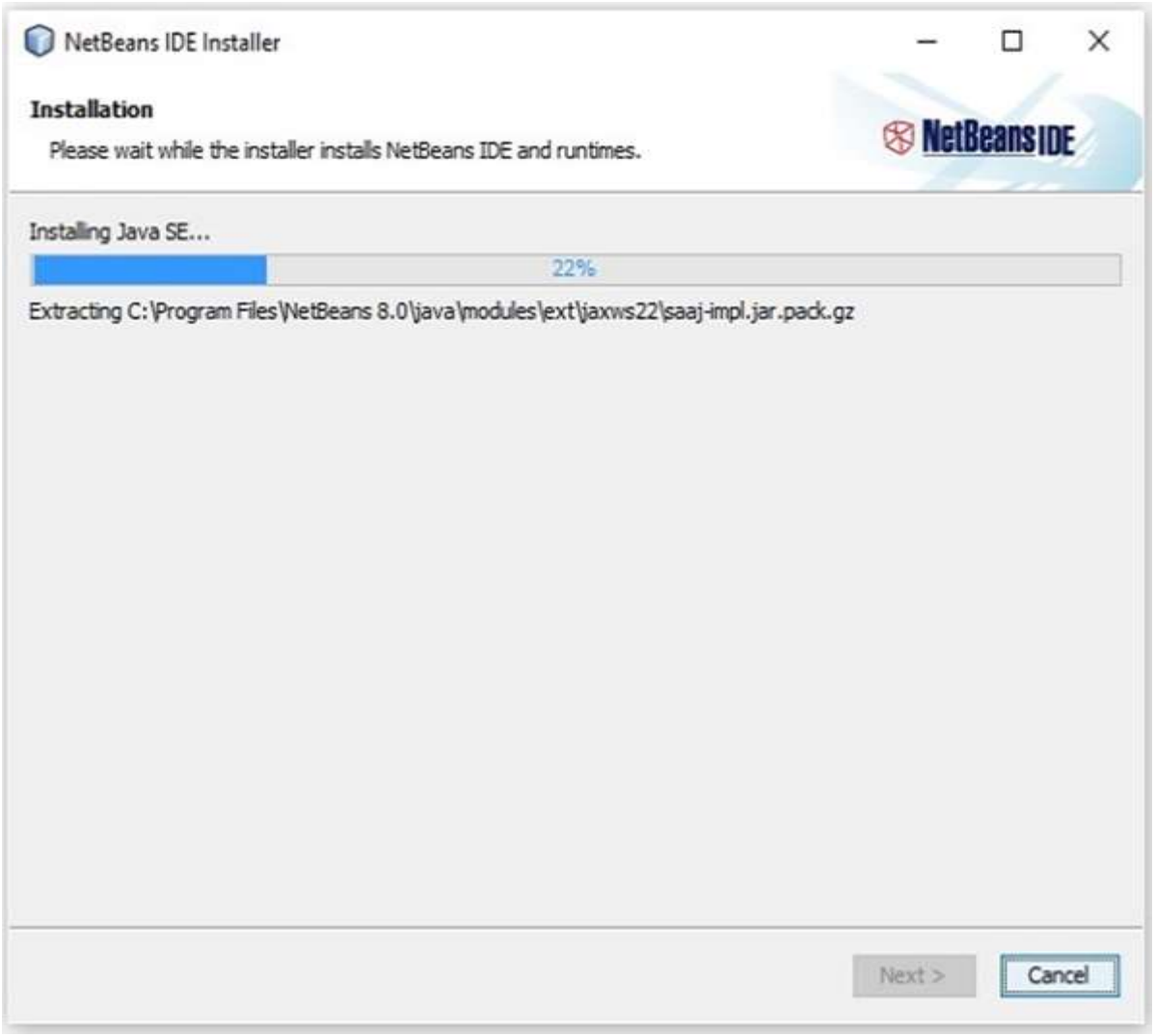
Step 8: Similarly, choose the destination directory for **Glassfish Server** installation. Browse through the Java Development Kit directory (now for Glassfish Reference) and then click **Next**.



Step 9: Check the **Check for Updates** box for automatic updates and click the **Install** button to start the installation.

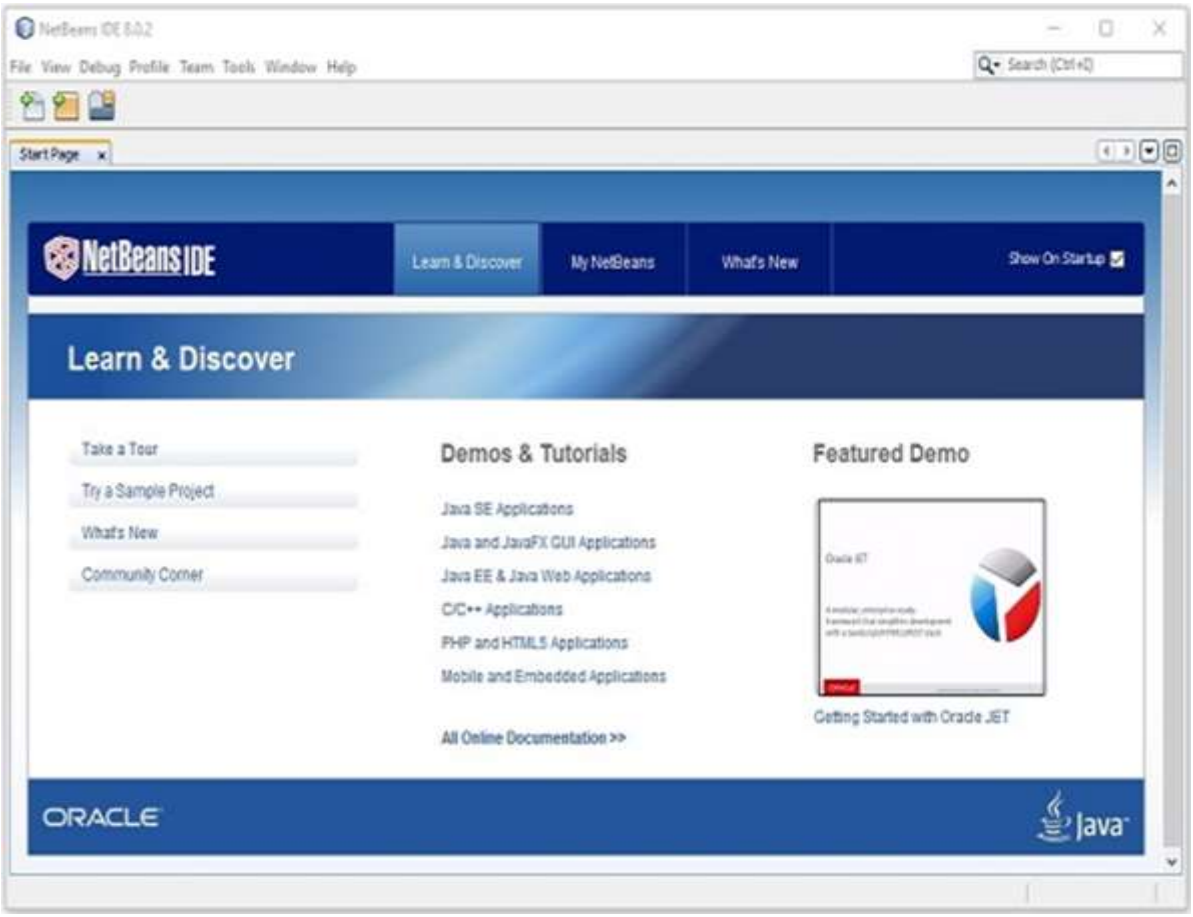


Step 10: This step starts the installation of NetBeans IDE 8.0 and it may take a while.

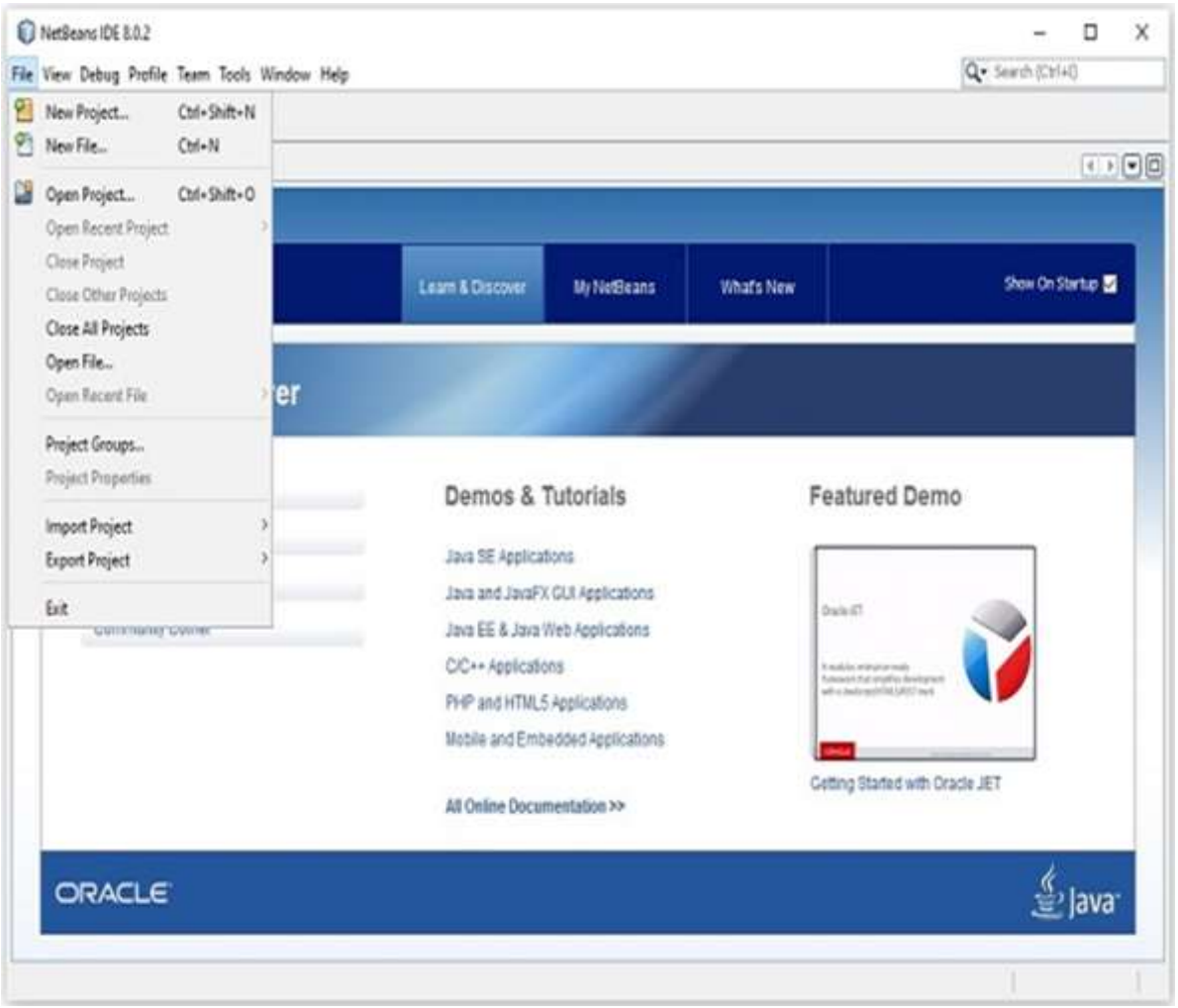


Step 11: Once the process is complete, click the **Finish** button to finish the installation.

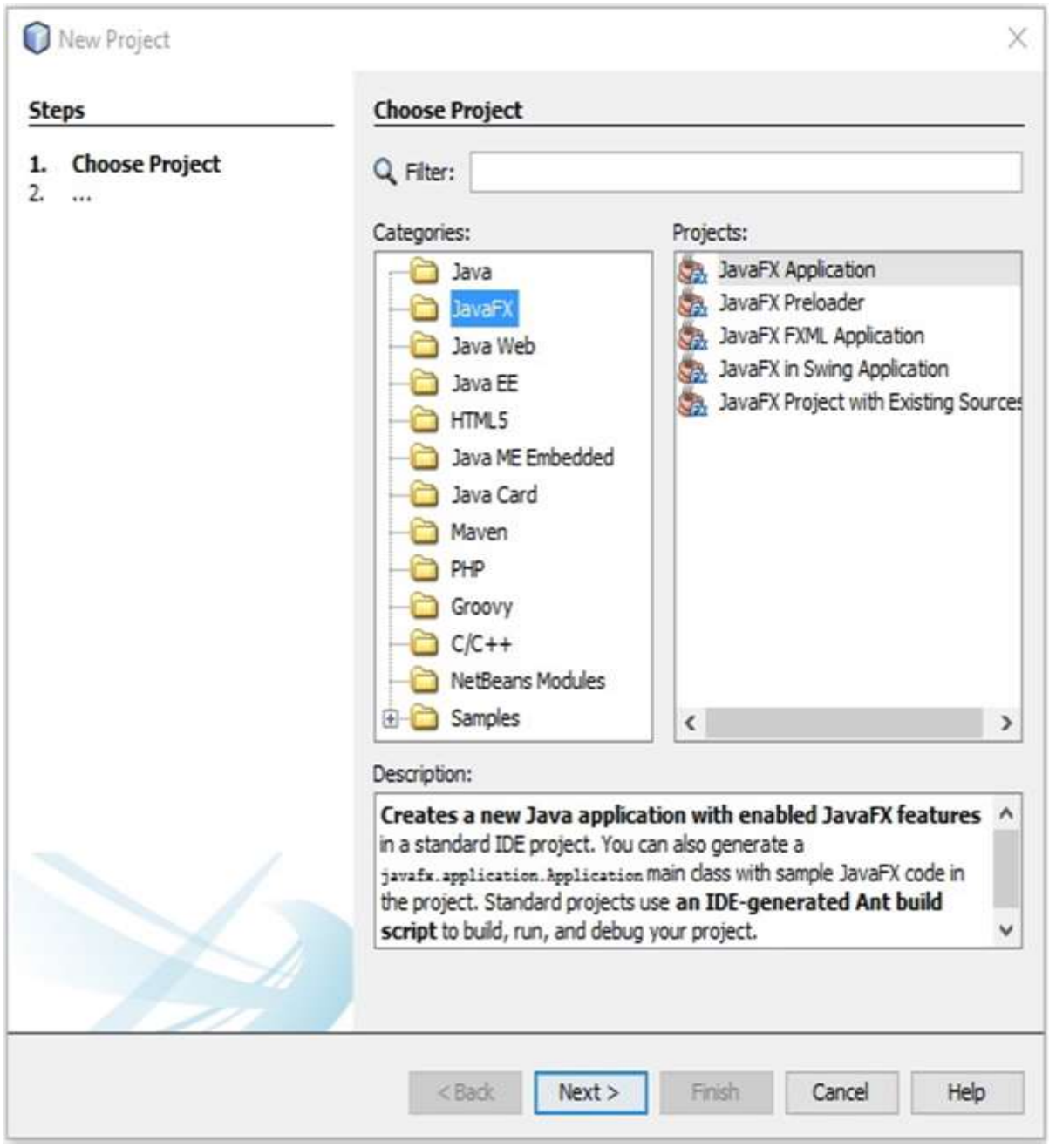
Step 12: Once you launch the NetBeans IDE, you will see the start page as shown in the following screenshot.



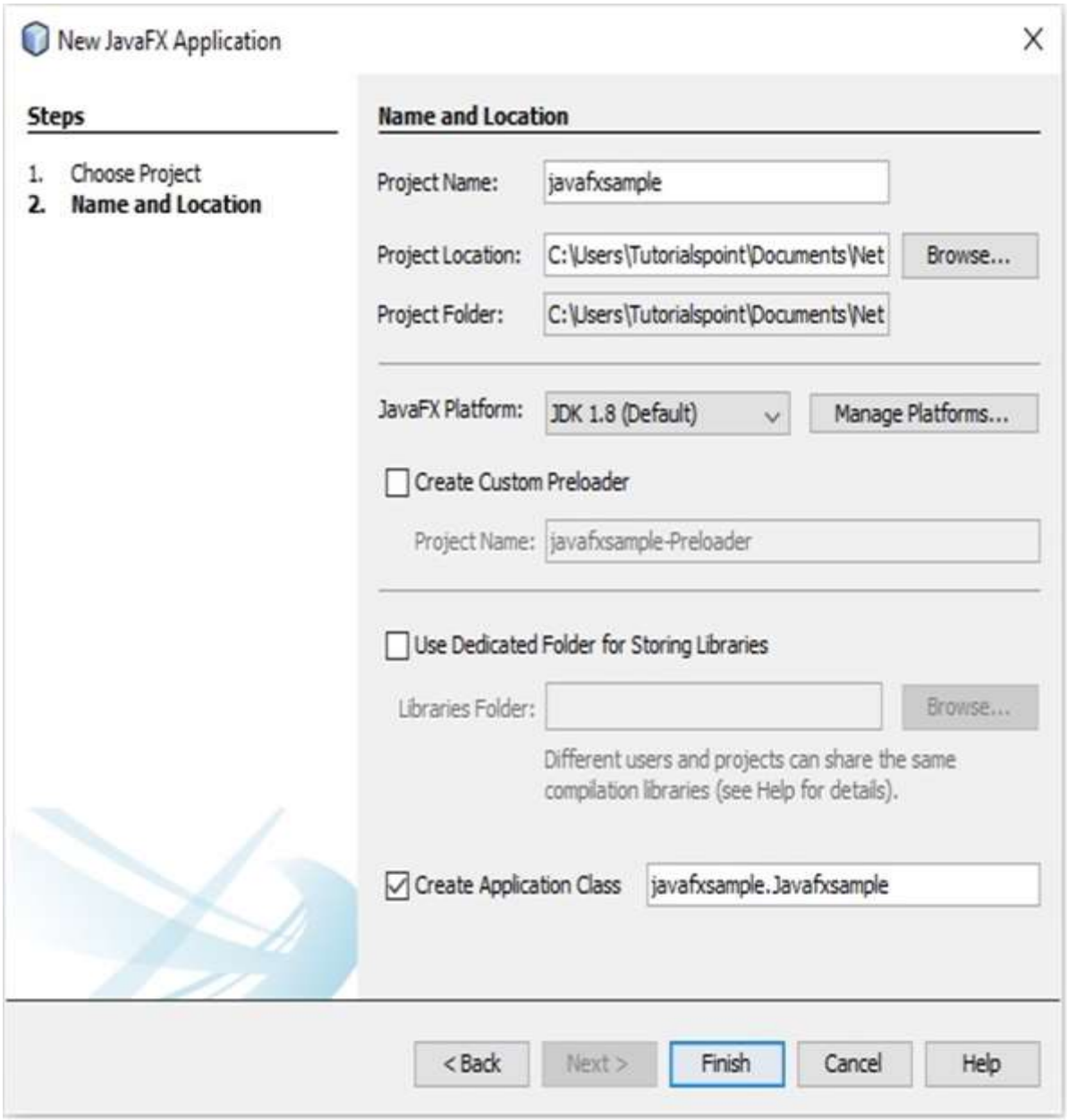
Step 13: In the file menu, select **New Project...** to open the New project wizard as shown in the following screenshot.



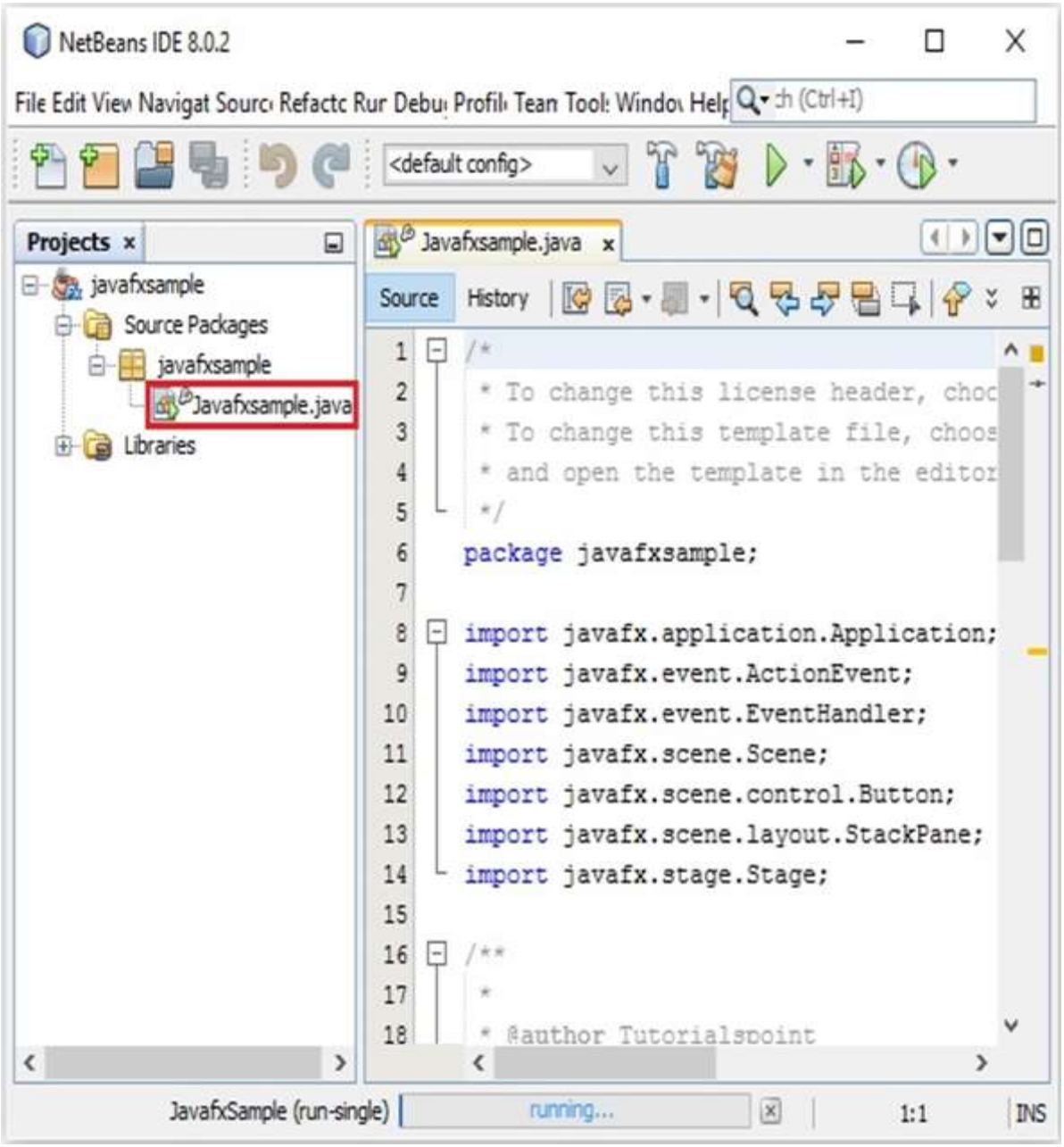
Step 14: In the **New Project** wizard, select **JavaFX** and click on **Next**. It starts creating a new JavaFX Application for you.



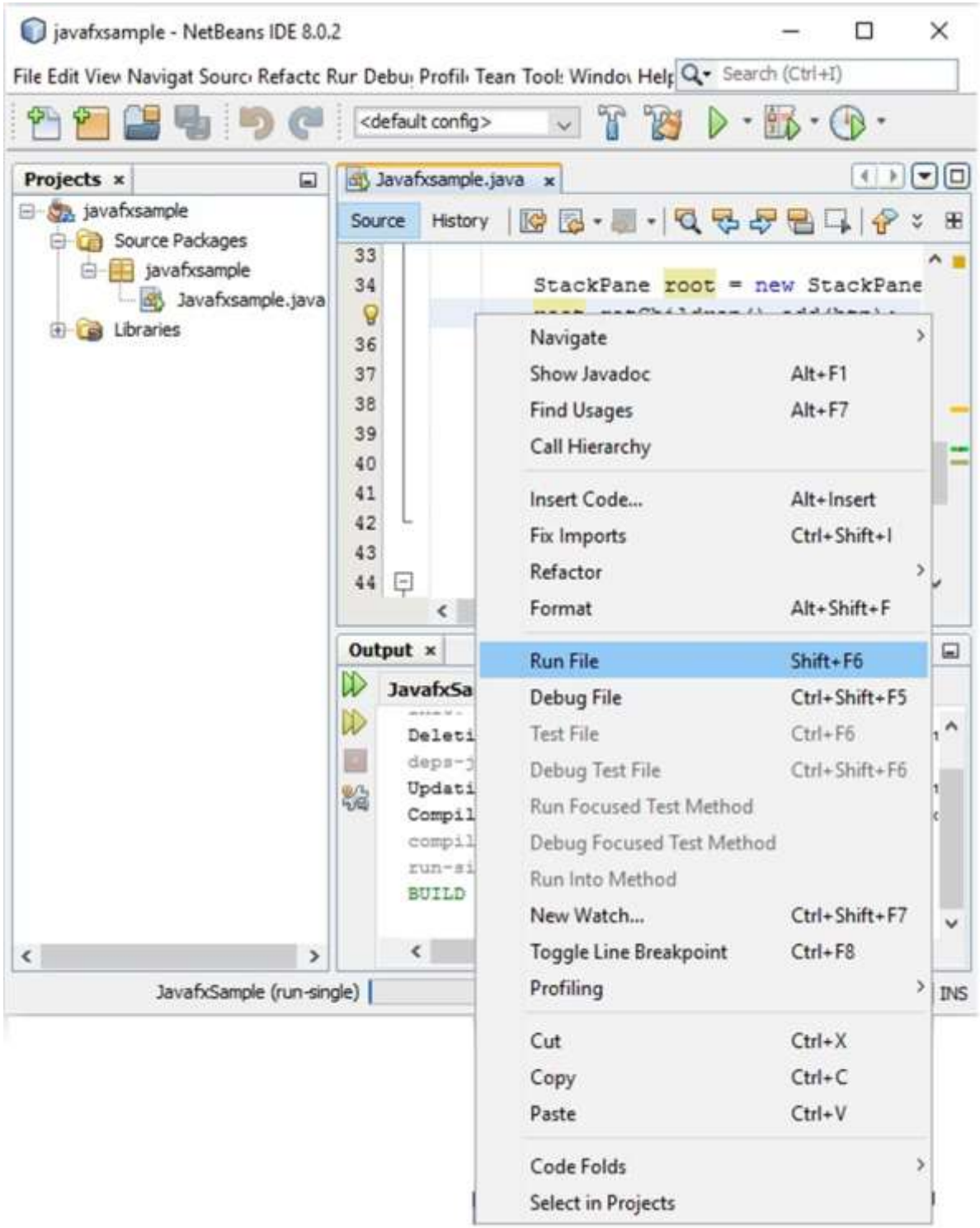
Step 15: Select the name of the project and location of the project in the **NewJavaFX Application** window and then click **Finish**. It creates a sample application with the given name.



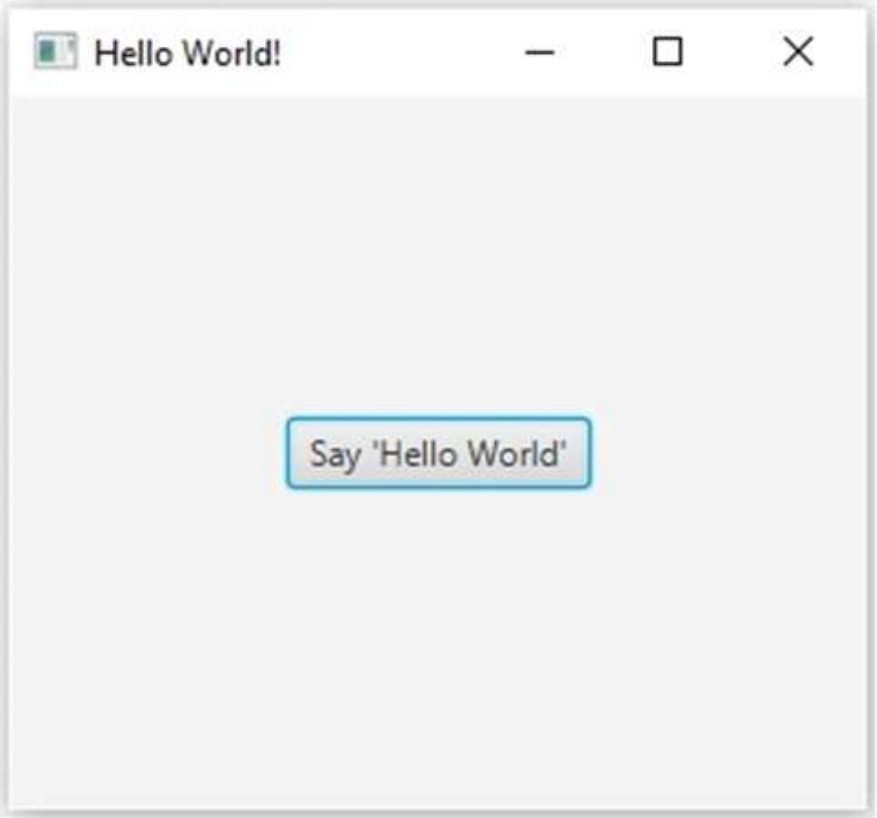
In this instance, an application with a name **javafxsample** is created. Within this application, the NetBeans IDE will generate a Java program with the name **Javafxsample.java**. As shown in the following screenshot, this program will be created inside NetBeans Source Packages → **javafxsample**.



Step 16: Right-click on the file and select **Run File** to run this code as shown in the following screenshot.



This automatically created program contains the code which generates a simple JavaFX window having a button with the label **Say 'Hello World'** in it. Every time you click on this button, the string **Hello World** will be displayed on the console as shown below.

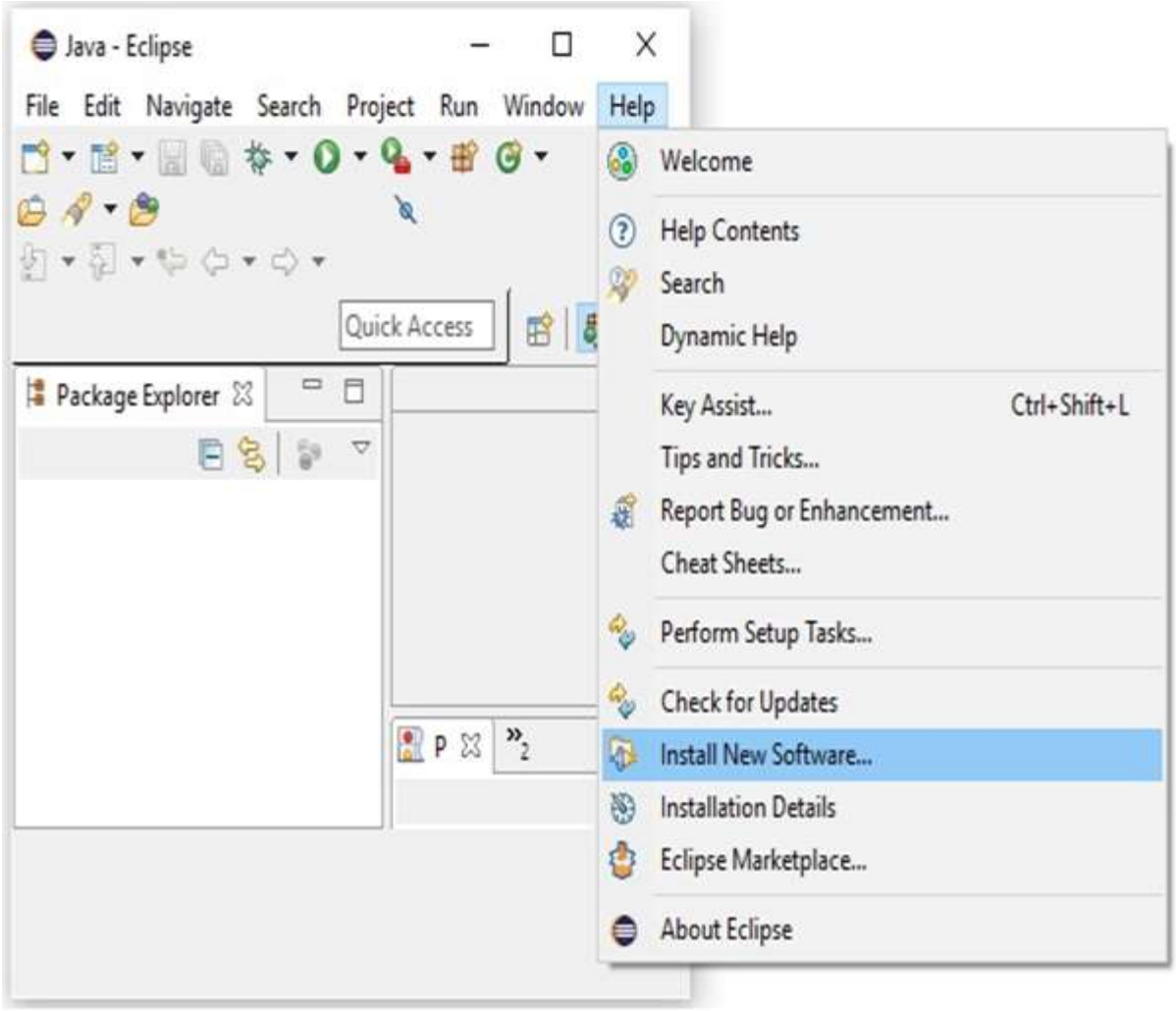


Installing JavaFX in Eclipse

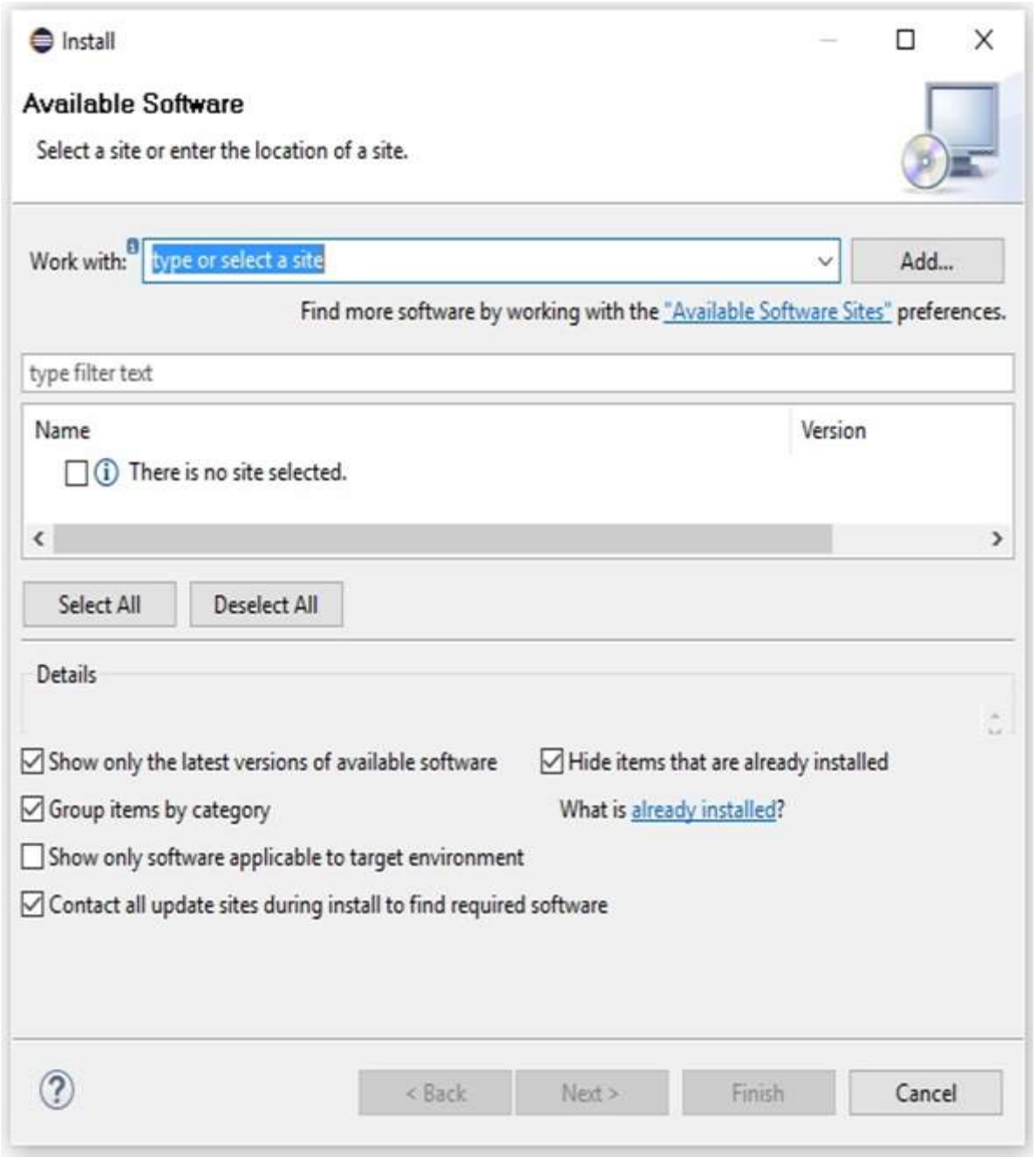
A plugin named **e(fx)clipse** is also available in JavaFX. You can use the following steps to set JavaFX in Eclipse. First of all, make sure that you have Eclipse in your system. If not, download and install Eclipse in your system.

Once Eclipse is installed, follow the steps given below to install **e(fx)clipse** in your system.

Step 1: Open Eclipse in the **Help** menu and select **Install New Software...** option as shown below.



Upon clicking, it will display the **Available Software** window, as shown in the following screenshot.

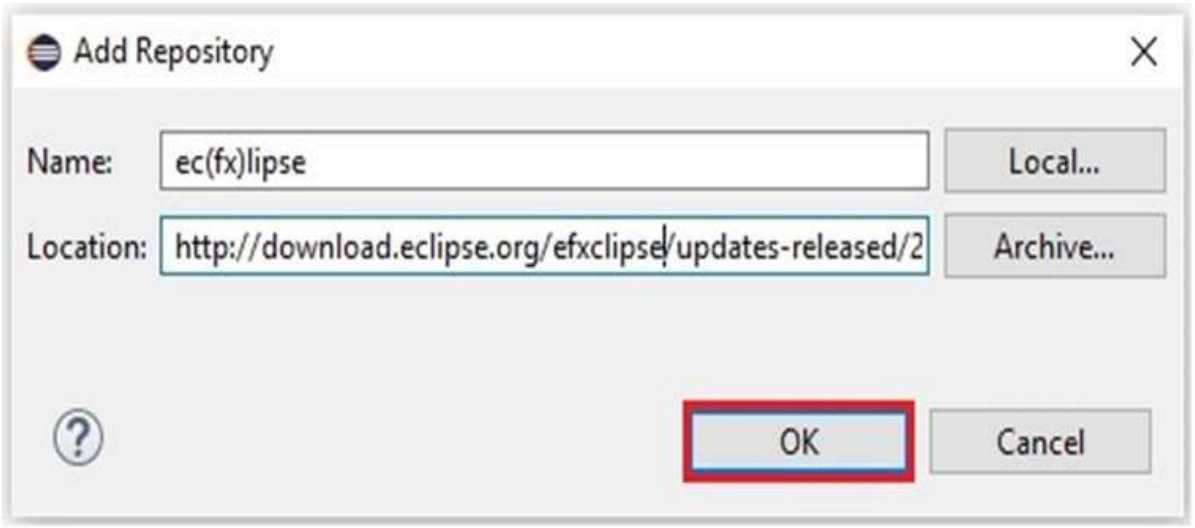


In the text box **Work with** of this window, you need to provide the link of the plugin for the required software.

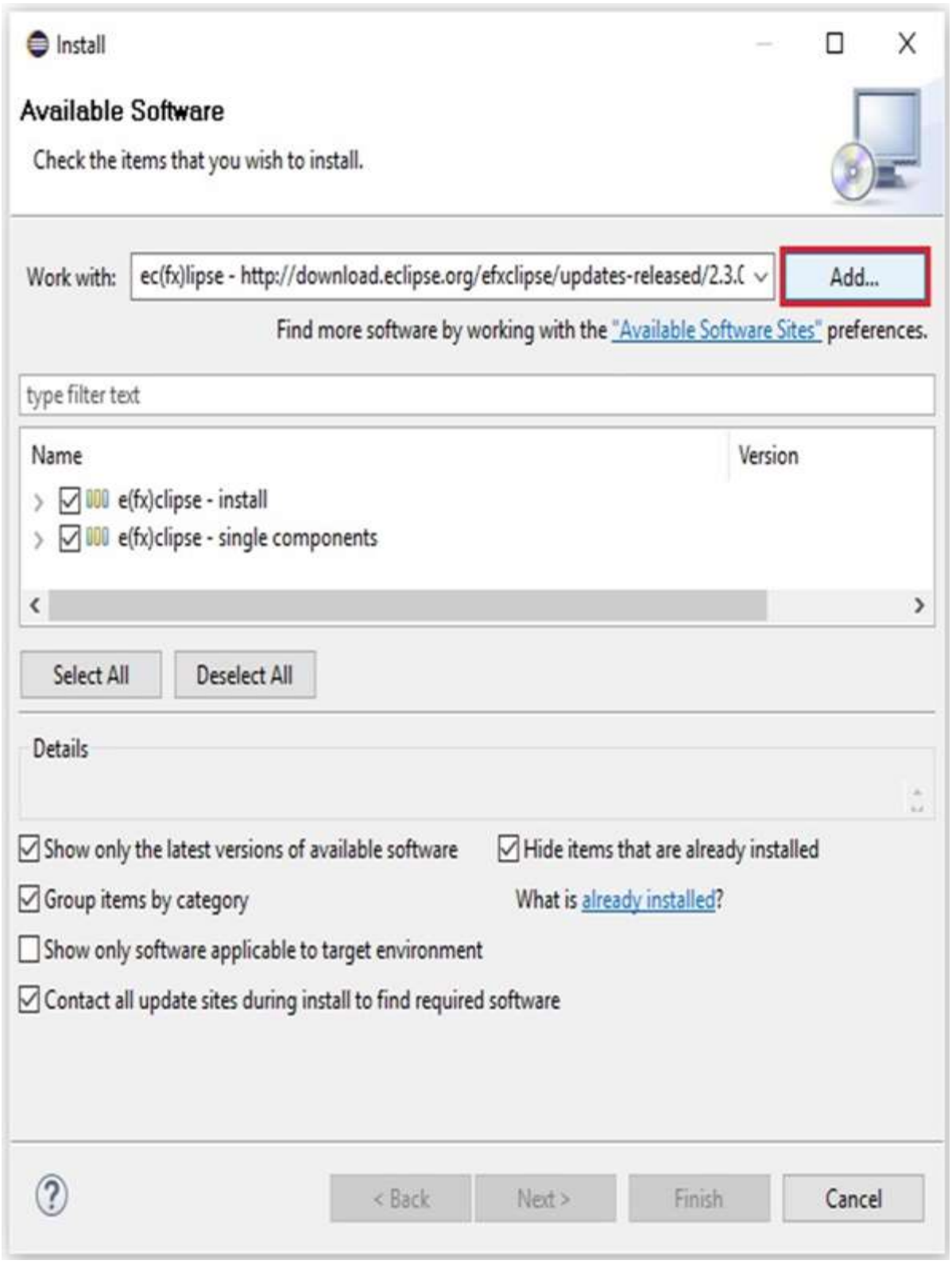
Step 2: Click the **Add...** button. Provide the name of the plugin as **e(fx)clipse**. Next, provide the following link as the location.

```
http://download.eclipse.org/efxclipse/updates-released/2.3.0/site
```

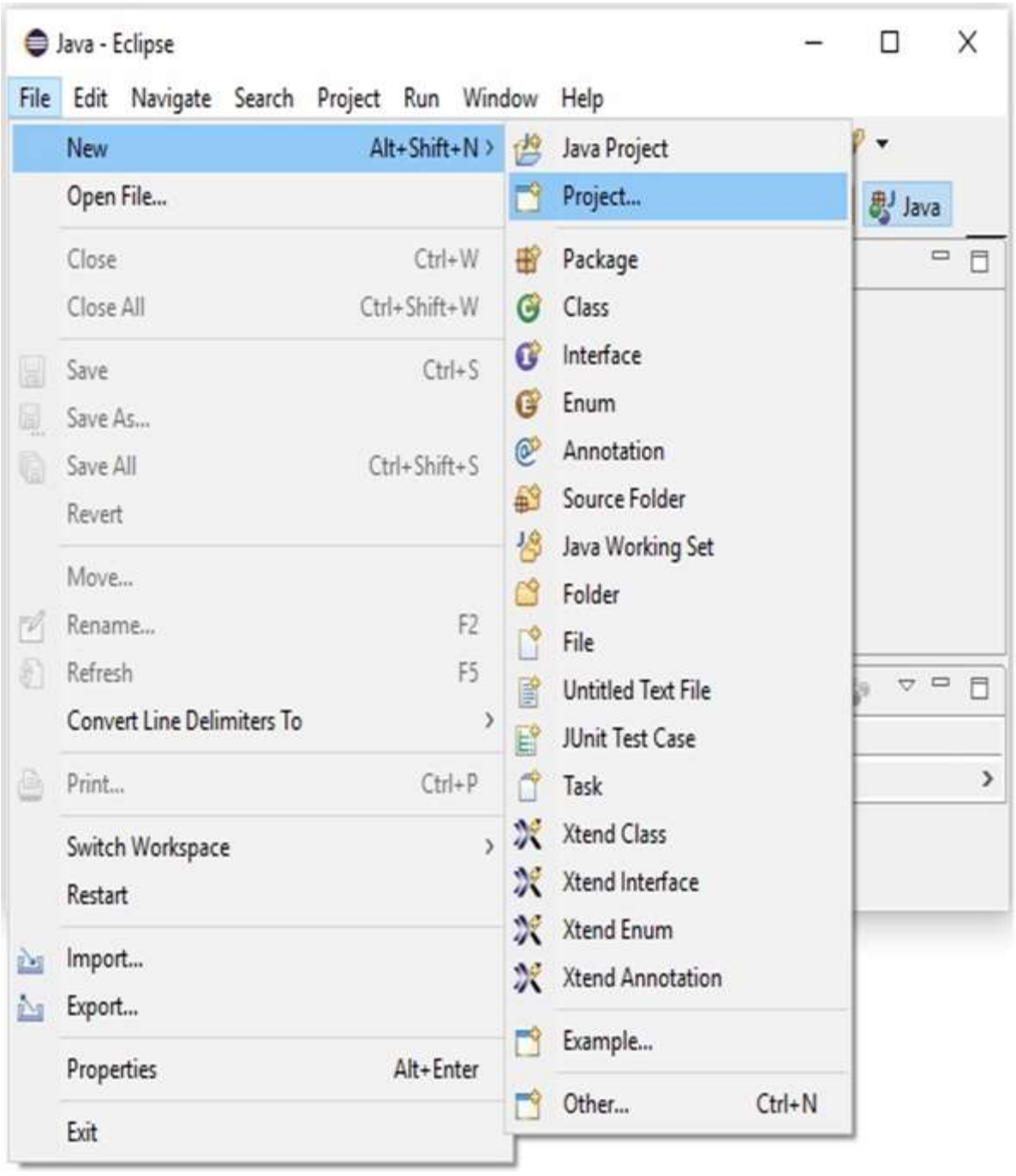
Step 3: After specifying the name and location of the plugin, click the **OK** button, as highlighted in the following screenshot.



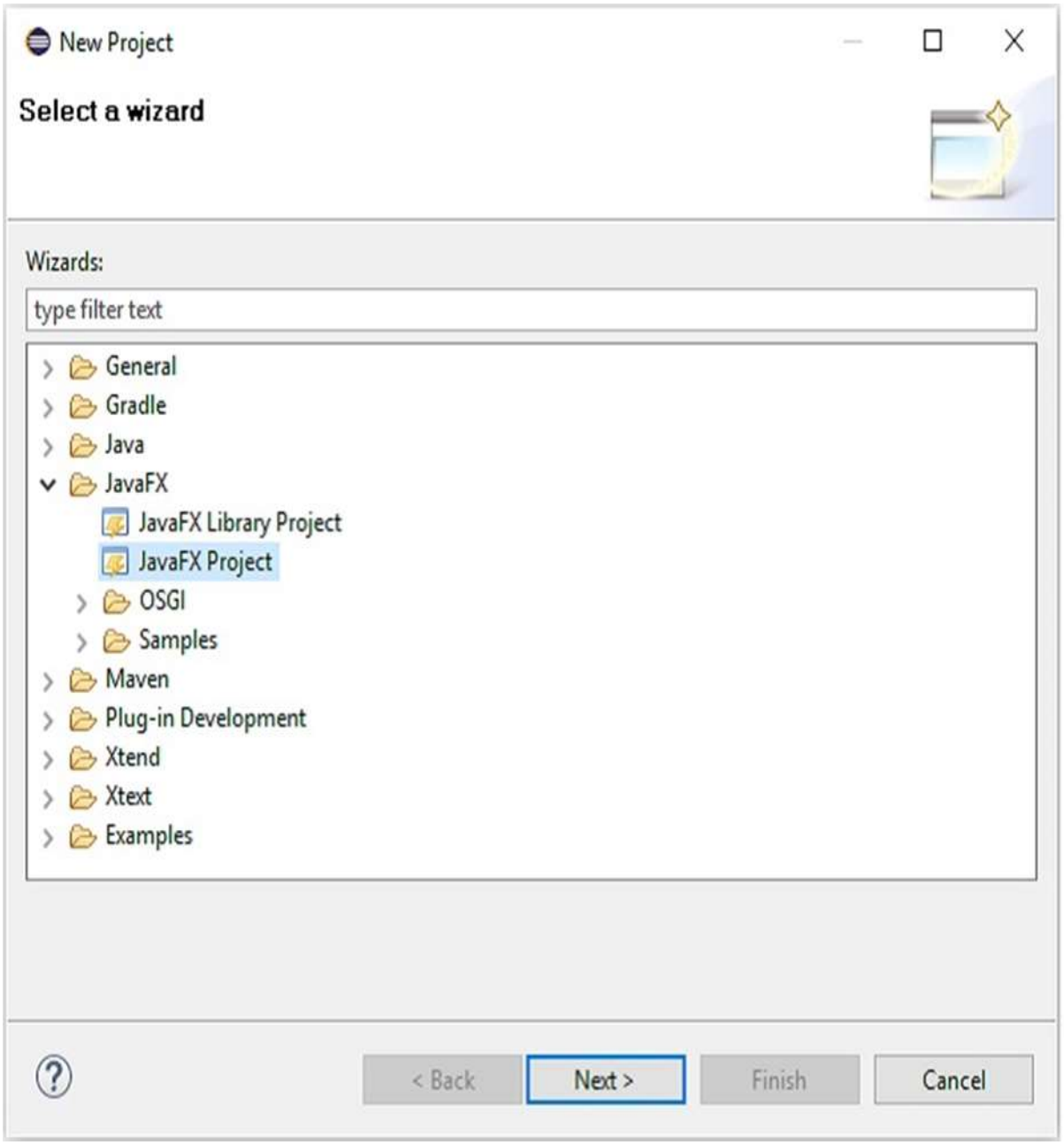
Step 4: Soon after you add the plugin, you will find two checkboxes for **e(fx)clipse – install** and **e(fx)clipse – single components**. Check both these checkboxes and click the **Add...** button as shown in the following screenshot.



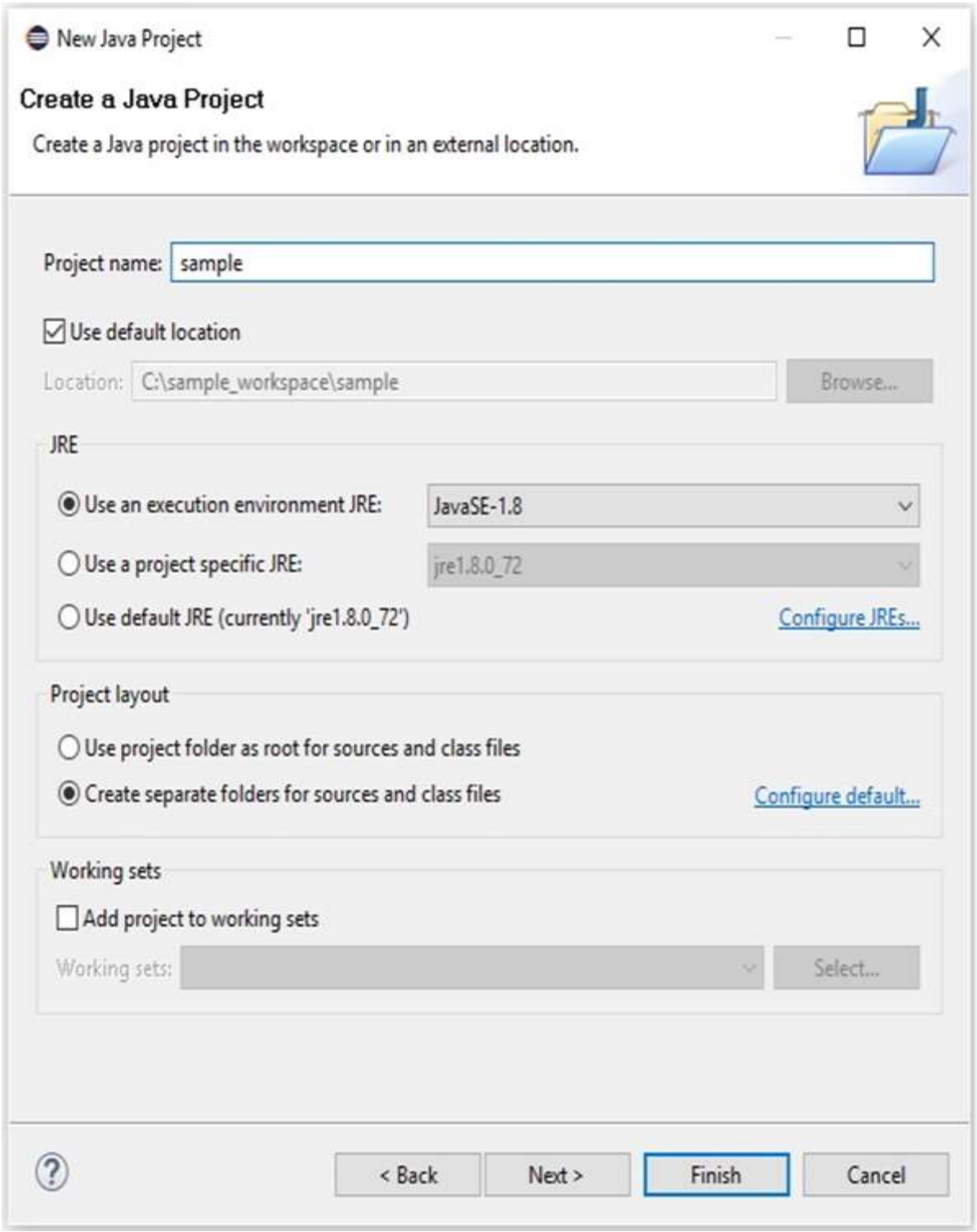
Step 5: Next, open your Eclipse IDE. Click the File menu and select Project as shown in the following screenshot.



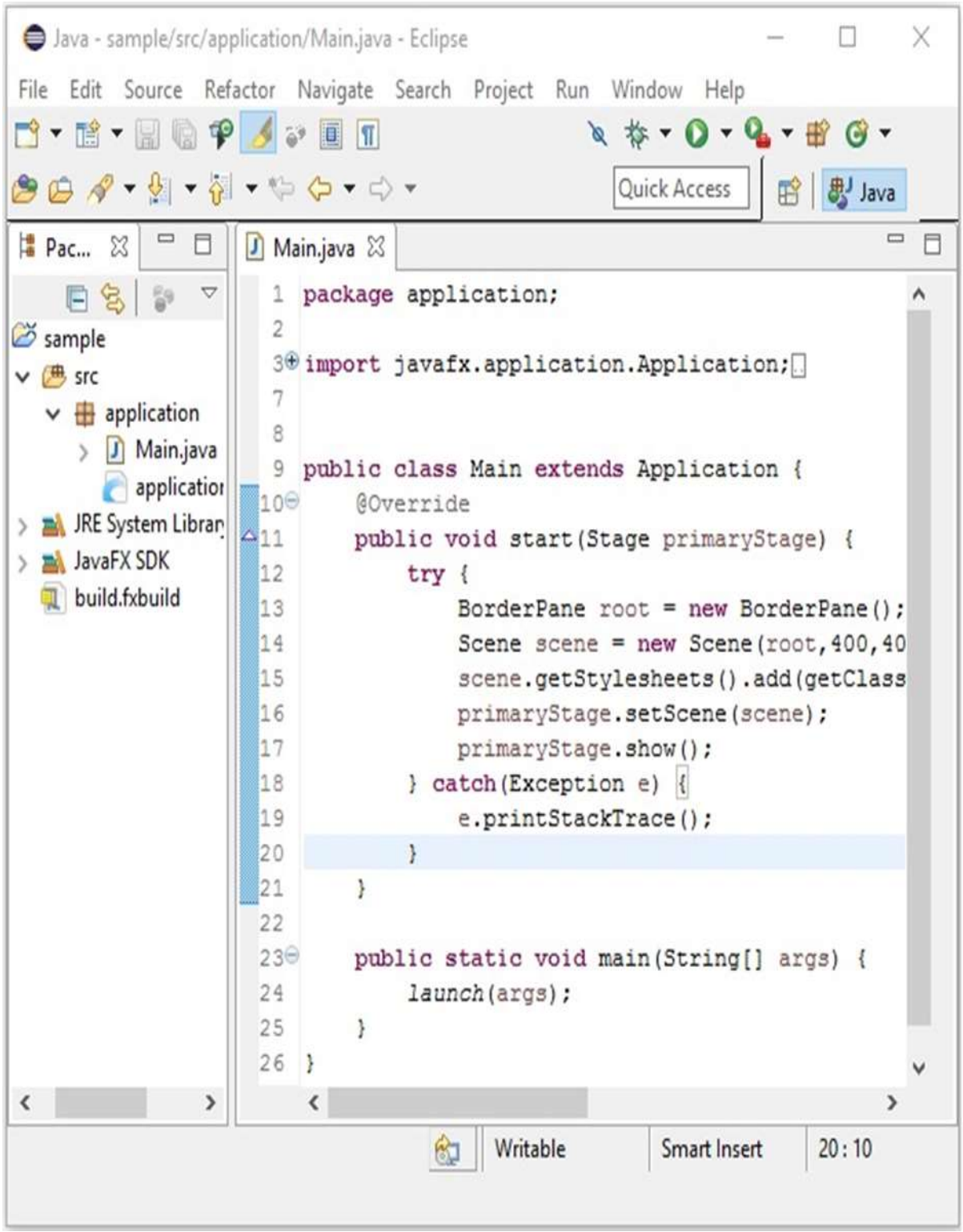
Step 6: Then, you will get a window where you can see a list of wizards provided by Eclipse to create a project. Expand the **JavaFX** wizard, select **JavaFX Project** and click the **Next** button as shown in the following screenshot.



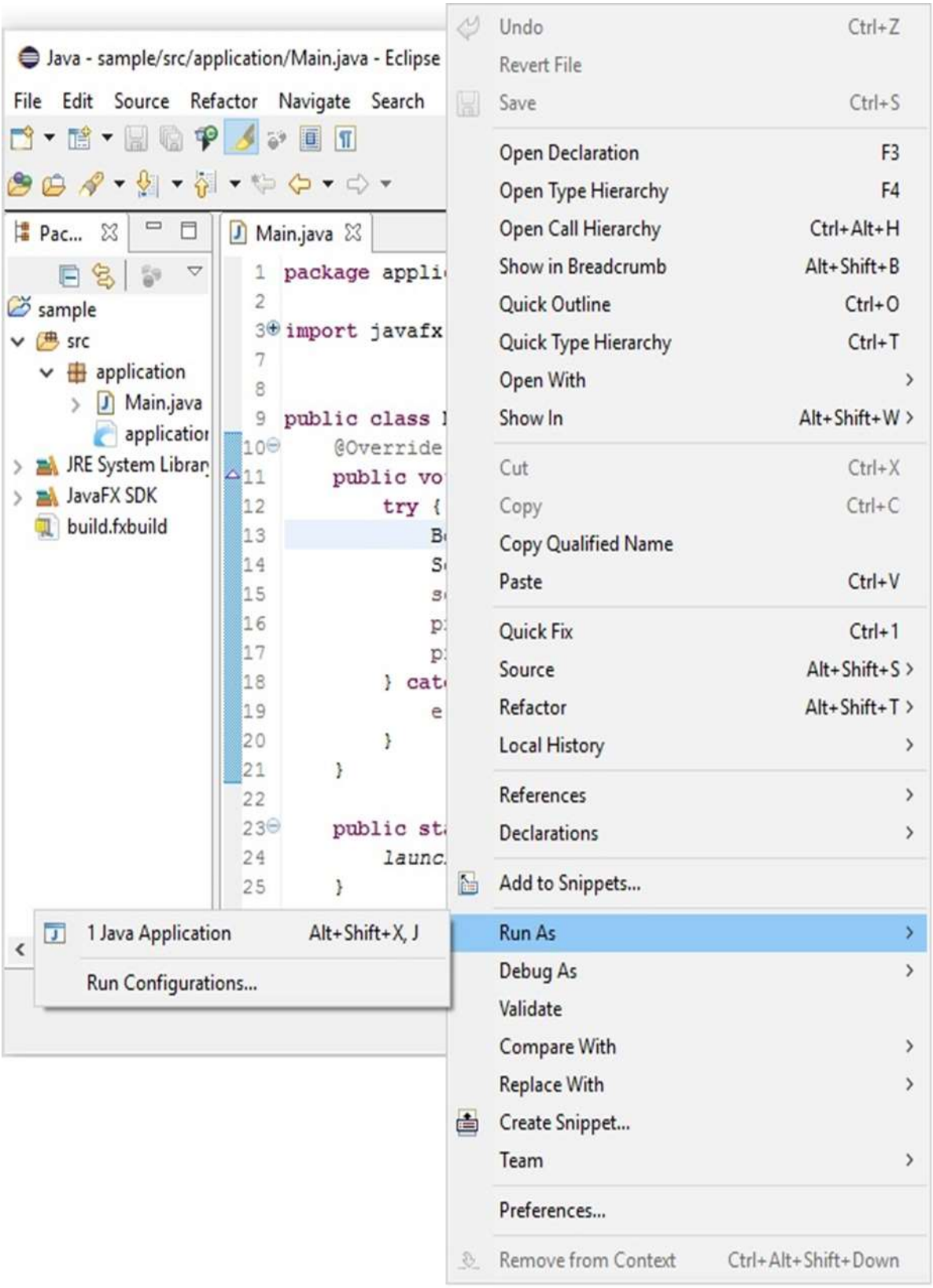
Step 7: On clicking **Next**, a New Project Wizard opens. Here, you can type the required project name and click **Finish.**



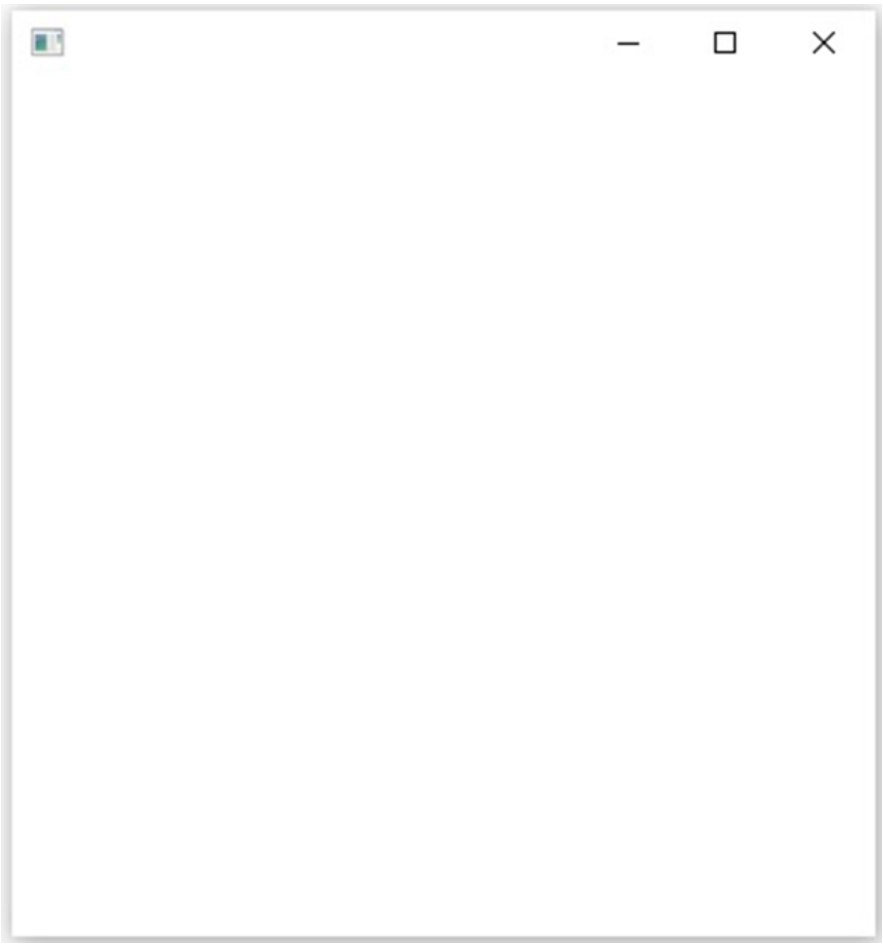
Step 8: On clicking Finish, an application is created with the given name (sample). In the sub-package named **application**, a program with the name **Main.java** is generated as shown below.



Step 9: This automatically generated program contains the code to generate an empty JavaFX window. Right-click on this file, select **Run As → Java Application** as shown in the following screenshot.



On executing this application, it gives you an empty JavaFX window as shown below.



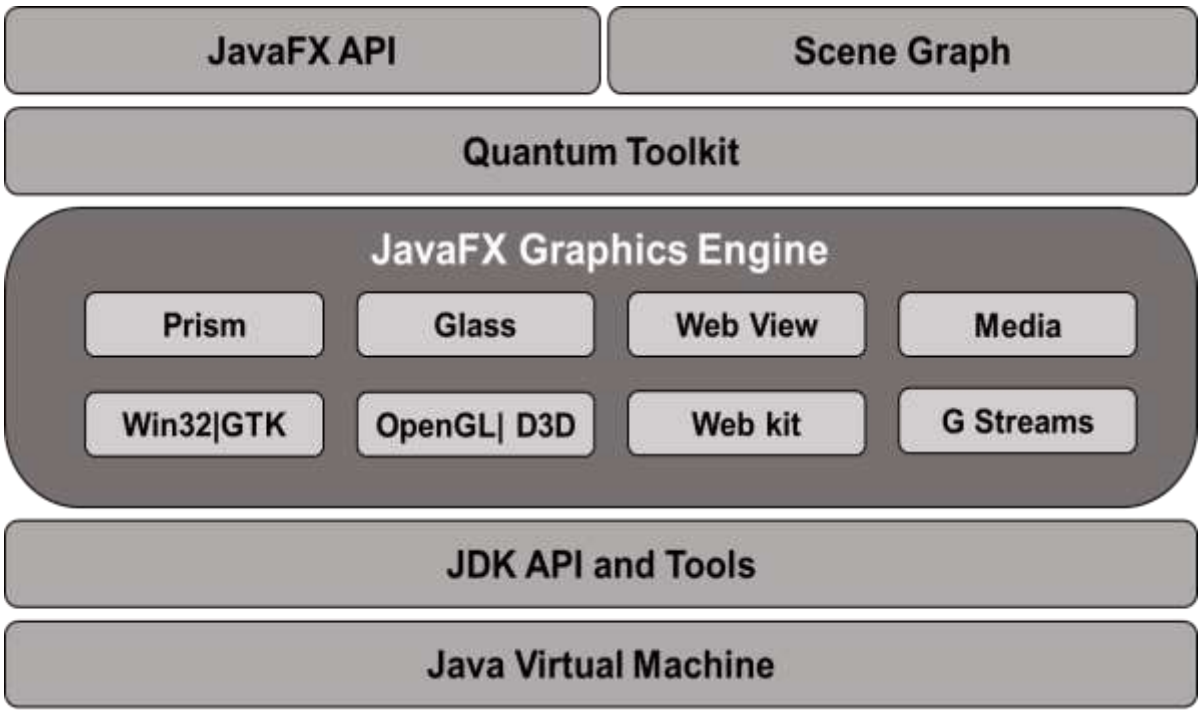
Note: We will discuss more about the code in the later chapters.

3. JavaFX – Architecture

JavaFX provides a complete API with a rich set of classes and interfaces to build GUI applications with rich graphics. The important packages of this API are –

- **javafx.animation:** Contains classes to add transition based animations such as fill, fade, rotate, scale and translation, to the JavaFX nodes.
- **javafx.application:** Contains a set of classes responsible for the JavaFX application life cycle.
- **javafx.css:** Contains classes to add CSS–like styling to JavaFX GUI applications.
- **javafx.event:** Contains classes and interfaces to deliver and handle JavaFX events.
- **javafx.geometry:** Contains classes to define 2D objects and perform operations on them.
- **javafx.stage:** This package holds the top level container classes for JavaFX application.
- **javafx.scene:** This package provides classes and interfaces to support the scene graph. In addition, it also provides sub-packages such as canvas, chart, control, effect, image, input, layout, media, paint, shape, text, transform, web, etc. There are several components that support this rich API of JavaFX.

The following illustration shows the architecture of JavaFX API. Here you can see the components that support JavaFX API.



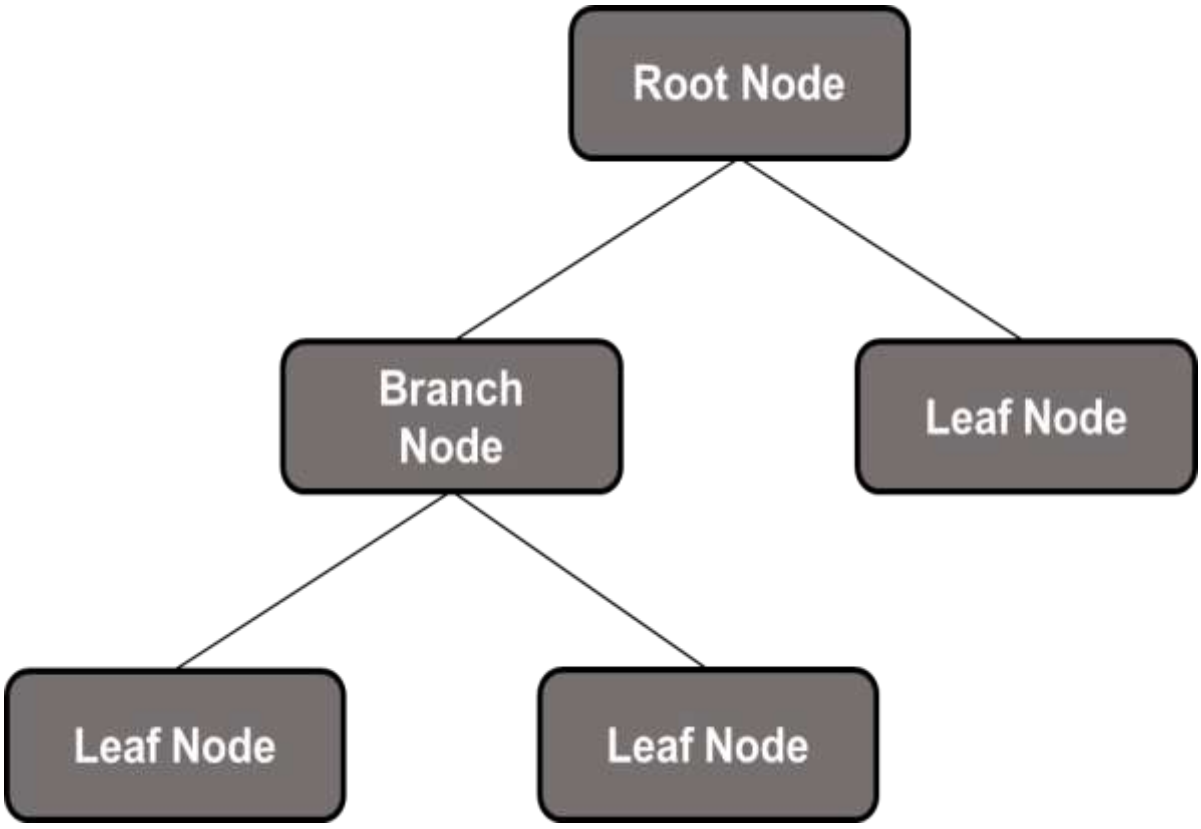
Scene Graph

In JavaFX, the GUI Applications were coded using a Scene Graph. A Scene Graph is the starting point of the construction of the GUI Application. It holds the (GUI) application primitives that are termed as nodes.

A node is a visual/graphical object and it may include —

- **Geometrical (Graphical) objects –** (2D and 3D) such as circle, rectangle, polygon, etc.
- **UI controls –** such as Button, Checkbox, Choice box, Text Area, etc.
- **Containers** – (layout panes) such as Border Pane, Grid Pane, Flow Pane, etc.
- **Media elements** – such as audio, video and image objects.

In general, a collection of nodes makes a scene graph. All these nodes are arranged in a hierarchical order as shown below.



Each node in the scene graph has a single parent, and the node which does not contain any parents is known as the **root node**.

In the same way, every node has one or more children, and the node without children is termed as **leaf node**; a node with children is termed as a **branch node**.

A node instance can be added to a scene graph only once. The nodes of a scene graph can have Effects, Opacity, Transforms, Event Handlers, Event Handlers, Application Specific States.

Prism

Prism is a **high performance hardware–accelerated graphical pipeline** that is used to render the graphics in JavaFX. It can render both 2-D and 3-D graphics.

To render graphics, a Prism uses –

- DirectX 9 on Windows XP and Vista.
- DirectX 11 on Windows 7.
- OpenGL on Mac and Linux, Embedded Systems.

In case the hardware support for graphics on the system is not sufficient, then Prism uses the software render path to process the graphics.

When used with a supported Graphic Card or GPU, it offers smoother graphics. Just in case the system does not support a graphic card, then Prism defaults to the software rendering stack (either of the above two).

GWT (Glass Windowing Toolkit)

As the name suggests, GWT provides services to manage Windows, Timers, Surfaces and Event Queues. GWT connects the JavaFX Platform to the Native Operating System.

Quantum Toolkit

It is an abstraction over the low-level components of Prism, Glass, Media Engine, and Web Engine. It ties Prism and GWT together and makes them available to JavaFX.

WebView

Using JavaFX, you can also embed HTML content in to a scene graph. **WebView** is the component of JavaFX which is used to process this content. It uses a technology called **Web Kit**, which is an internal open-source web browser engine. This component supports different web technologies like HTML5, CSS, JavaScript, DOM and SVG.

Using WebView, you can –

- Render HTML content from local or remote URL.
- Support history and provide Back and Forward navigation.
- Reload the content.
- Apply effects to the web component.
- Edit the HTML content.
- Execute JavaScript commands.
- Handle events.

In general, using WebView, you can control web content from Java.

Media Engine

The **JavaFX media engine** is based on an open-source engine known as a **Streamer**. This media engine supports the playback of video and audio content.

The JavaFX media engine provides support for audio for the following file formats:

Audio	• MP3 • AIFF • WAV
Video	• FLV

The package **javafx.scene.media** contains the classes and interfaces to provide media functionality in JavaFX. It is provided in the form of three components, which are –

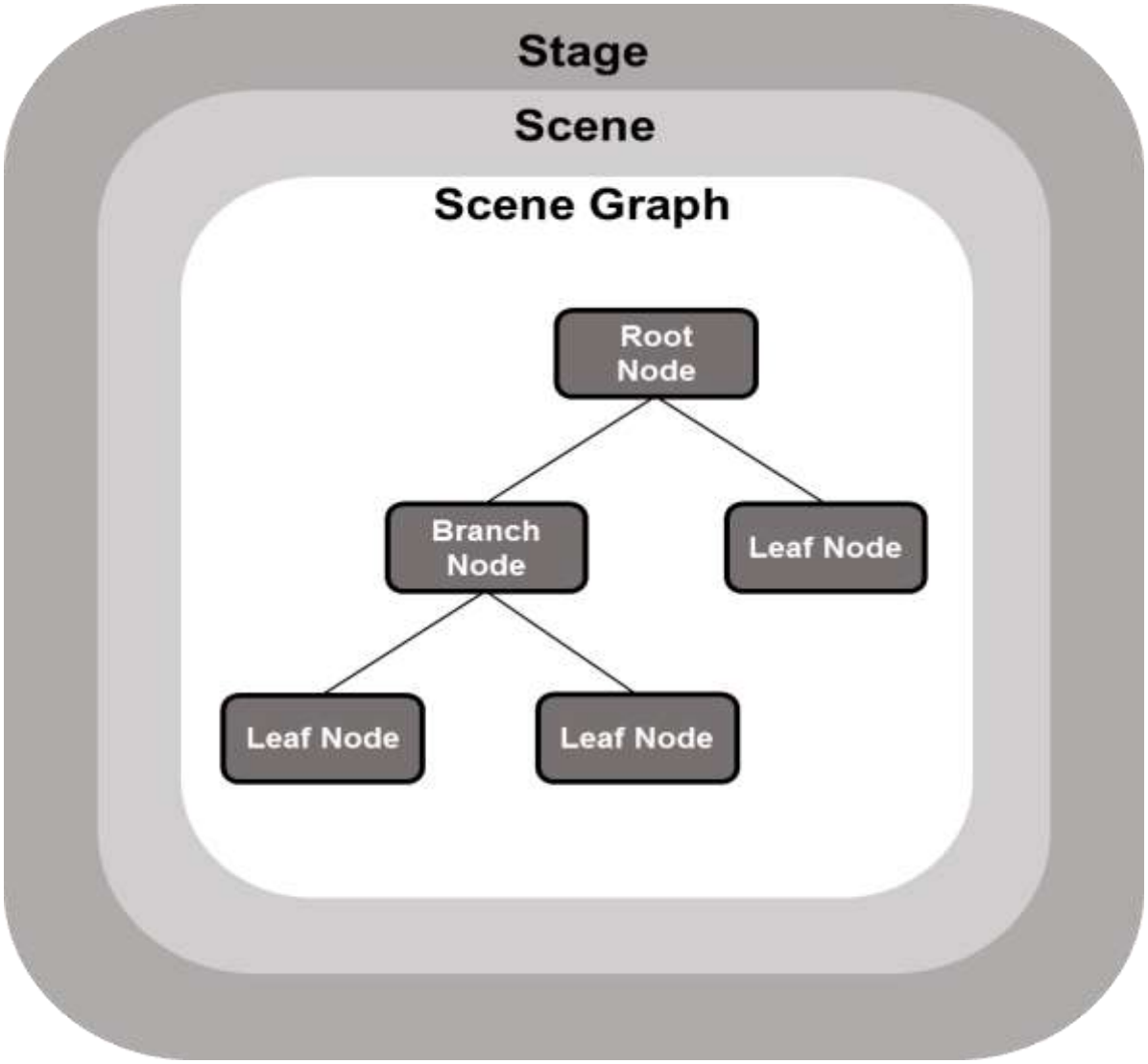
- Media Object: This represents a media file.
- Media Player: To play media content.
- Media View: To display media.

4. JavaFX – Application

In this chapter, we will discuss the structure of a JavaFX application in detail and also learn to create a JavaFX application with an example.

JavaFX Application Structure

In general, a JavaFX application will have three major components namely **Stage, Scene** and **Nodes** as shown in the following diagram.



Stage

A stage (a window) contains all the objects of a JavaFX application. It is represented by **Stage** class of the package **javafx.stage.** The primary stage is created by the platform itself. The created stage object is passed as an argument to the **start()** method of the **Application** class (explained in the next section).

A stage has two parameters determining its position namely **Width** and **Height**. It is divided as Content Area and Decorations (Title Bar and Borders).

There are five types of stages available –

- Decorated
- Undecorated
- Transparent
- Unified
- Utility

You have to call the **show()** method to display the contents of a stage.

Scene

A scene represents the physical contents of a JavaFX application. It contains all the contents of a scene graph. The class **Scene** of the package **javafx.scene** represents the scene object. At an instance, the scene object is added to only one stage.

You can create a scene by instantiating the Scene Class. You can opt for the size of the scene by passing its dimensions (height and width) along with the **root node** to its constructor.

Scene Graph and Nodes

A **scene graph** is a tree-like data structure (hierarchical) representing the contents of a scene. In contrast, a **node** is a visual/graphical object of a scene graph.

A node may include —

- Geometrical (Graphical) objects (2D and 3D) such as – Circle, Rectangle, Polygon, etc.
- UI Controls such as – Button, Checkbox, Choice Box, Text Area, etc.
- Containers (Layout Panes) such as Border Pane, Grid Pane, Flow Pane, etc.
- Media elements such as Audio, Video and Image Objects.

The **Node** Class of the package **javafx.scene** represents a node in JavaFX, this class is the super class of all the nodes.

As discussed earlier a node is of three types –

- **Root Node:** The first Scene Graph is known as the Root node.
- **Branch Node/Parent Node:** The node with child nodes are known as branch/parent nodes. The abstract class named **Parent** of the package **javafx.scene** is the base class of all the parent nodes, and those parent nodes will be of the following types –
    - **Group:** A group node is a collective node that contains a list of children nodes. Whenever the group node is rendered, all its child nodes are

rendered in order. Any transformation, effect state applied on the group will be applied to all the child nodes.

- **Region:** It is the base class of all the JavaFX Node based UI Controls, such as Chart, Pane and Control.
- **WebView:** This node manages the web engine and displays its contents.

- **Leaf Node:** The node without child nodes is known as the leaf node. For example, Rectangle, Ellipse, Box, ImageView, MediaView are examples of leaf nodes.

It is mandatory to pass the root node to the scene graph. If the Group is passed as root, all the nodes will be clipped to the scene and any alteration in the size of the scene will not affect the layout of the scene.

Creating a JavaFX Application

To create a JavaFX application, you need to instantiate the Application class and implement its abstract method **start().** In this method, we will write the code for the JavaFX Application.

Application Class

The **Application** class of the package **javafx.application** is the entry point of the application in JavaFX. To create a JavaFX application, you need to inherit this class and implement its abstract method **start().** In this method, you need to write the entire code for the JavaFX graphics.

In the **main** method, you have to launch the application using the **launch()** method. This method internally calls the **start()** method of the Application class as shown in the following program.

```
public class JavafxSample extends Application {


    @Override
    public void start(Stage primaryStage) throws Exception {
    /*
     Code for JavaFX application.
     (Stage, scene, scene graph)
    */
    }
    public static void main(String args[]){
        launch(args);
   }
}
```

Within the **start()** method, in order to create a typical JavaFX application, you need to follow the steps given below –

- Prepare a scene graph with the required nodes.
- Prepare a Scene with the required dimensions and add the scene graph (root node of the scene graph) to it.
- Prepare a stage and add the scene to the stage and display the contents of the stage.

Preparing the Scene Graph

As per your application, you need to prepare a scene graph with required nodes. Since the root node is the first node, you need to create a root node. As a root node, you can choose from the **Group**, **Region** or **WebView**.

Group: A Group node is represented by the class named **Group** which belongs to the package **javafx.scene**, you can create a Group node by instantiating this class as shown below.

```
Group root = new Group();
```

The **getChildren()** method of the **Group** class gives you an object of the **ObservableList** class which holds the nodes. We can retrieve this object and add nodes to it as shown below.

```
//Retrieving the observable list object
ObservableList list = root.getChildren();


//Setting the text object as a node
list.add(NodeObject);
```

We can also add Node objects to the group, just by passing them to the **Group** class and to its constructor at the time of instantiation, as shown below.

```
Group root = new Group(NodeObject);
```

Region: It is the Base class of all the JavaFX Node-based UI Controls, such as –

- **Chart:** This class is the base class of all the charts and it belongs to the package **javafx.scene.chart**.

  This class has two sub classes, which are – **PieChart** and **XYChart**. These two in turn have subclasses such as **AreaChart**, **BarChart, BubbleChart,** etc**.** used to draw different types of XY-Plane Charts in JavaFX.

  You can use these classes to embed charts in your application.

- **Pane:** A Pane is the base class of all the layout panes such as **AnchorPane, BorderPane, DialogPane,** etc**.** This class belong to a package that is called as – **javafx.scene.layout**.

  You can use these classes to insert predefined layouts in your application.

- **Control:** It is the base class of the User Interface controls such as **Accordion, ButtonBar, ChoiceBox, ComboBoxBase, HTMLEditor,** etc**.** This class belongs to the package **javafx.scene.control**.

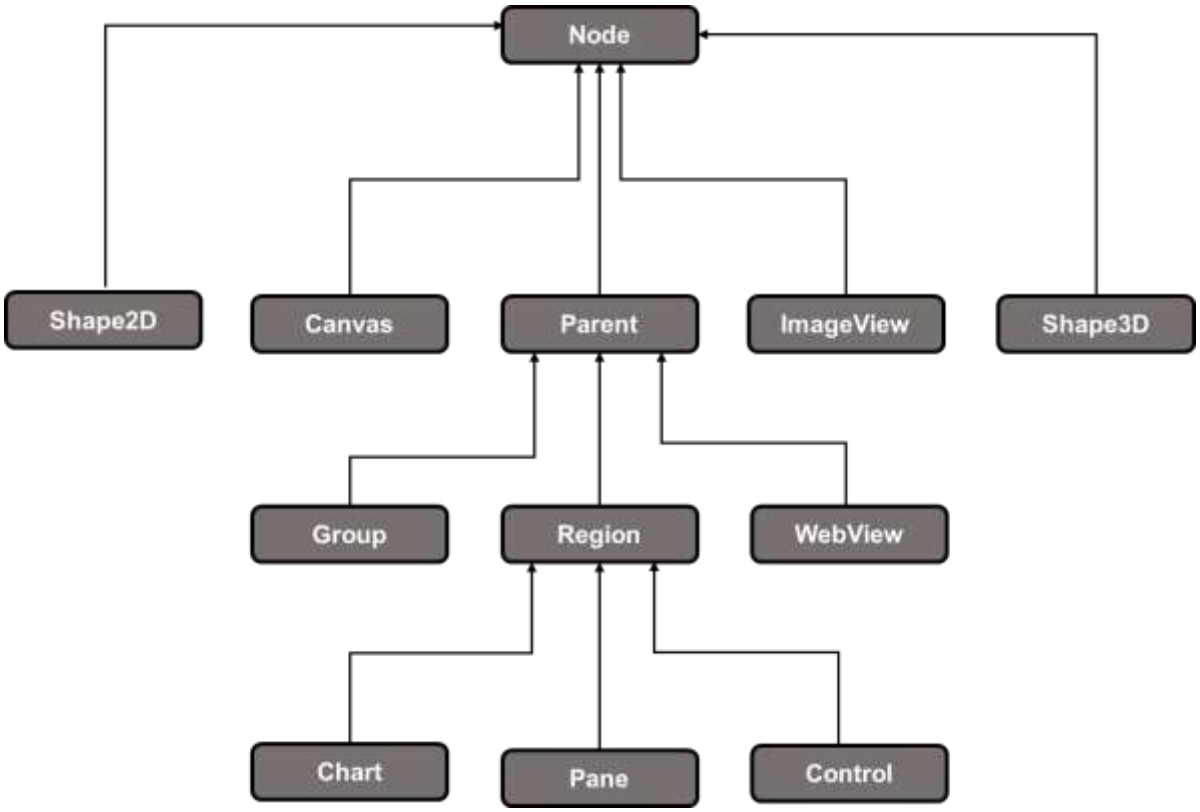
  You can use these classes to insert various UI elements in your application.

In a Group, you can instantiate any of the above-mentioned classes and use them as root nodes, as shown in the following program.

```
//Creating a Stack Pane
StackPane pane = new StackPane();


//Adding text area to the pane
ObservableList list = pane.getChildren();
list.add(NodeObject);
```

WebView: This node manages the web engine and displays its contents.

Following is a diagram representing the node class hierarchy of JavaFX.



Preparing the Scene

A JavaFX scene is represented by the **Scene** class of the package **javafx.scene**. You can create a Scene by instantiating this class as shown in the following cod block.

While instantiating, it is mandatory to pass the root object to the constructor of the scene class.

```
Scene scene = new Scene(root);
```

You can also pass two parameters of double type representing the height and width of the scene as shown below.

```
Scene scene = new Scene(root, 600, 300);
```

Preparing the Stage

This is the container of any JavaFX application and it provides a window for the application. It is represented by the **Stage** class of the package **javafx.stage**. An object of this class is passed as a parameter of the **start()** method of the **Application** class.

Using this object, you can perform various operations on the stage. Primarily you can perform the following –

- Set the title for the stage using the method **setTitle()**.
- Attach the scene object to the stage using the **setScene()** method.
- Display the contents of the scene using the **show()** method as shown below.

```
//Setting the title to Stage.
primaryStage.setTitle("Sample application");

//Setting the scene to Stage
primaryStage.setScene(scene);

//Displaying the stage
primaryStage.show();
```

Lifecycle of JavaFX Application

The JavaFX Application class has three life cycle methods, which are —

- **start():** The entry point method where the JavaFX graphics code is to be written.
- **stop():** An empty method which can be overridden, here you can write the logic to stop the application.

- **init():** An empty method which can be overridden, but you cannot create stage or scene in this method.

In addition to these, it provides a static method named **launch()** to launch JavaFX application.

Since the **launch()** method is static, you need to call it from a static context (main generally). Whenever a JavaFX application is launched, the following actions will be carried out (in the same order).

- An instance of the application class is created.
- **Init()** method is called.
- The **start()** method is called.
- The launcher waits for the application to finish and calls the **stop()** method.

Terminating the JavaFX Application

When the last window of the application is closed, the JavaFX application is terminated implicitly. You can turn this behavior off by passing the Boolean value "False" to the static method **setImplicitExit()** (should be called from a static context).

You can terminate a JavaFX application explicitly using the methods **Platform.exit()** or **System.exit(**int**)**.

Example 1 – Creating an Empty Window (Stage)

This section teaches you how to create a JavaFX sample application which displays an empty window. Following are the steps:

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application** and implement the **start()** method of this class as follows.

```
public class JavafxSample extends Application {


    @Override
    public void start(Stage primaryStage) throws Exception {


    }


}
```

Step 2: Creating a Group Object

In the **start()** method creates a group object by instantiating the class named **Group**, which belongs to the package **javafx.scene**, as follows.

```
Group root = new Group();
```

Step 3: Creating a Scene Object

Create a Scene by instantiating the class named **Scene** which belongs to the package **javafx.scene.** To this class, pass the Group object (**root**), created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen along with the object of the Group class as follows.

```
Scene scene = new Scene(group,600, 300);
```

Step 4: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object which is passed to the start method of the scene class, as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as shown below.

```
primaryStage.setTitle("Sample Application");
```

Step 5: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using this method as shown below.

```
primaryStage.setScene(scene);
```

Step 6: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 7: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows.

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

Example

The following program generates an empty JavaFX window. Save this code in a file with the name **JavafxSample.java**

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

public class JavafxSample extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {

      //creating a Group object
      Group group = new Group();

      //Creating a Scene by passing the group object, height and width
      Scene scene = new Scene(group ,600, 300);

      //setting color to the scene
      scene.setFill(Color.BROWN);

      //Setting the title to Stage.
      primaryStage.setTitle("Sample Application");

      //Adding the scene to Stage
      primaryStage.setScene(scene);

      //Displaying the contents of the stage
      primaryStage.show();

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac JavafxSample.java
java JavafxSample
```

On executing, the above program generates a JavaFX window as shown below.

Example 2 – Drawing a Straight Line

In the previous example, we have seen how to create an empty stage, now in this example let us try to draw a straight line using the JavaFX library.

Following are the steps:

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application** and implement the **start()** method of this class as follows.

```
public class DrawingLine extends Application {
```

```
    @Override
    public void start(Stage primaryStage) throws Exception {
    }
}
```

Step 2: Creating a Line

You can create a line in JavaFX by instantiating the class named **Line** which belongs to a package **javafx.scene.shape**, instantiate this class as follows.

```
//Creating a line object
Line line = new Line();
```

Step 3: Setting Properties to the Line

Specify the coordinates to draw the line on an X-Y plane by setting the properties **startX, startY, endX** and **endY**, using their respective setter methods as shown in the following code block.

```
line.setStartX(100.0);
line.setStartY(150.0);
line.setEndX(500.0);
line.setEndY(150.0);
```

Step 4: Creating a Group Object

In the start() method create a group object by instantiating the class named Group, which belongs to the package javafx.scene.

Pass the Line (node) object, created in the previous step, as a parameter to the constructor of the Group class, in order to add it to the group as follows –

```
Group root = new Group(line);
```

Step 5: Creating a Scene Object

Create a Scene by instantiating the class named **Scene** which belongs to the package **javafx.scene.** To this class, pass the Group object (**root**) that was created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 6: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object which is passed to the start method of the scene class, as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 7: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using this method as follows.

```
primaryStage.setScene(scene);
```

Step 8: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 9: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows.

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

Example

The following program shows how to generate a straight line using JavaFX. Save this code in a file with the name **JavafxSample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.shape.Line;
import javafx.stage.Stage;

public class DrawingLine extends Application{
```

```
   @Override
   public void start(Stage stage) {

      //Creating a line object
      Line line = new Line();

      //Setting the properties to a line
      line.setStartX(100.0);
      line.setStartY(150.0);
      line.setEndX(500.0);
      line.setEndY(150.0);

      //Creating a Group
      Group root = new Group(line);

      //Creating a Scene
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the scene
      stage.setTitle("Sample application");

      //Adding the scene to the stage
      stage.setScene(scene);

      //Displaying the contents of a scene
      stage.show();

    }

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

}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac DrawingLine.java
java DrawingLine
```

On executing, the above program generates a JavaFX window displaying a straight line as shown below.

Example 3 – Displaying Text

We can also embed text in JavaFX scene. This example shows how to embed text in JavaFX.

Following are the steps:

Step 1: Creating a Class

Create a Java Class and inherit the **Application** class of the package **javafx.application** and implement the **start()** method of this class as follows.

```
public class DrawingLine extends Application {


    @Override
```

```
    public void start(Stage primaryStage) throws Exception {
    }
}
```

Step 2: Embedding Text

You can embed text in to a JavaFX scene by instantiating the class named **Text** which belongs to a package **javafx.scene.shape**, instantiate this class.

To the constructor of this class pass the text to be embedded in String format as follows.

```
//Creating a Text object
Text text = new Text("Welcome to Tutorialspoint");
```

Step 3: Setting the Font

You can set font to the text using the **setFont()** method of the **Text** class. This method accepts a font object as parameters. Set the font of the given text to 45 as shown below.

```
//Setting font to the text
text.setFont(new Font(45));
```

Step 4: Setting the Position of the Text

You can set the position of the text on the X-Y plane by setting the X,Y coordinates using the respective setter methods **setX()** and **setY()** as follows.

```
//setting the position of the text
text.setX(50);
text.setY(150);
```

Step 5: Setting Properties to the Line

Specify the coordinates to draw the line on an X-Y plane by setting the properties **startX, startY, endX** and **endY**, using their respective setter methods as shown in the following code block.

```
line.setStartX(100.0);
line.setStartY(150.0);
line.setEndX(500.0);
line.setEndY(150.0);
```

Step 6: Creating a Group Object

In the **start()** method, create a group object by instantiating the class named **Group**, which belongs to the package **javafx.scene**.

Pass the Text (node) object, created in the previous step, as a parameter to the constructor of the Group class, in order to add it to the group as follows –

```
Group root = new Group(text);
```

Step 7: Creating a Scene Object

Create a Scene by instantiating the class named **Scene** which belongs to the package **javafx.scene.** To this class, pass the Group object (**root**), created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 8: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object which is passed to the start method of the scene class, as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as shown below.

```
primaryStage.setTitle("Sample Application");
```

Step 9: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using this method as follows.

```
primaryStage.setScene(scene);
```

Step 10: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 11: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows.

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

```
}
```

Example

Following is the program to display text using JavaFX. Save this code in a file with name **DisplayingText.java**.

```
import javafx.application.Application;
import javafx.collections.ObservableList;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.text.Font;
import javafx.scene.text.Text;

public class DisplayingText extends Application {
   @Override
   public void start(Stage stage) {

      //Creating a Text object
      Text text = new Text();

      //Setting font to the text
      text.setFont(new Font(45));

      //setting the position of the text
      text.setX(50);
      text.setY(150);

      //Setting the text to be added.
      text.setText("Welcome to Tutorialspoint");

      //Creating a Group object
      Group root = new Group();

      //Retrieving the observable list object
      ObservableList list = root.getChildren();
```

```
      //Setting the text object as a node to the group object
      list.add(text);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Sample Application");

      //Adding scene to the stage
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac DisplayingText.java
java DisplayingText
```

On executing, the above program generates a JavaFX window displaying text as shown below.

5. JavaFX – 2D Shapes

In the previous chapter, we have seen the basic application of JavaFX, where we learnt how to create an empty window and how to draw a line on an XY plane of JavaFX. In addition to the line, we can also draw several other 2D shapes.

2D Shape

In general, a 2D shape is a geometrical figure that can be drawn on the XY plane, these include Line, Rectangle, Circle, etc.

Using the JavaFX library, you can draw –

- Predefined shapes such as Line, Rectangle, Circle, Ellipse, Polygon, Polyline, Cubic Curve, Quad Curve, Arc.
- Path elements such as MoveTO Path Element, Line, Horizontal Line, Vertical Line, Cubic Curve, Quadratic Curve, Arc.
- In addition to these, you can also draw a 2D shape by parsing SVG path.

Each of the above mentioned 2D shape is represented by a class and all these classes belongs to the package **javafx.scene.shape**. The class named **Shape** is the base class of all the 2-Dimensional shapes in JavaFX.

Creating a 2D Shape

To create a chart, you need to –

- Instantiate the respective class of the required shape.
- Set the properties of the shape.
- Add the shape object to the group.

Instantiating the Respective Class

To create a 2 Dimensional shape, first of all you need to instantiate its respective class.

For example, if you want to create a line you need to instantiate the class named **Line** as follows –

```
Line line = new Line();
```

Setting the Properties of the Shape

After instantiating the class, you need to set the properties for the shape using the setter methods.

For example, to draw a line you need to pass its x and y coordinates of the start point and end point of the line. You can specify these values using their respective setter methods as follows —

```
//Setting the Properties of the Line
line.setStartX(150.0f);
line.setStartY(140.0f);
line.setEndX(450.0f);
line.setEndY(140.0f);
```

Adding the Shape Object to the Group

Finally, you need to add the object of the shape to the group by passing it as a parameter of the constructor as shown below.

```
//Creating a Group object
Group root = new Group(line);
```

The following table gives you the list of various shapes (classes) provided by JavaFX.

S. No.	Shape and Description
1	**Line:** A line is a geometrical structure joining two points. The **Line** class of the package **javafx.scene.shape** represents a line in the XY plane. It has four properties namely – • **startX:** The x coordinate of the start point of the line. • **startY:** The y coordinate of the start point of the line. • **endX:** The x coordinate of the end point of the line. • **endY:** The Y coordinate of the end point of the line.
2	**Rectangle:** In general, a rectangle is a four-sided polygon that has two pairs of parallel and concurrent sides with all interior angles as right angles. In JavaFX, a Rectangle is represented by a class named **Rectangle.** This class belongs to the package **javafx.scene.shape**. This class has 4 properties of the double datatype namely – • **x:** The x coordinate of the start point (upper left) of the rectangle. • **y:** The y coordinate of the start point (upper left) of the rectangle. • **width:** The width of the rectangle. • **height:** The height of the rectangle.

3	**Rounded Rectangle:** In JavaFX, you can draw a rectangle either with sharp edges or with arched edges and the one with arched edges is known as a rounded rectangle.
4	**Circle:** A circle is a line forming a closed loop, every point on which is a fixed distance from a centre point. In JavaFX, a circle is represented by a class named **Circle.** This class belongs to the package **javafx.scene.shape**. This class has 3 properties of the double datatype namely – • **centerX:** The x coordinate of the center of a circle. • **centerY:** The y coordinate of the center of a circle. • **radius:** The radius of the circle in pixels.
5	**Ellipse:** An ellipse is defined by two points, each called a focus. If any point on the ellipse is taken, the sum of the distances to the focus points is constant. The size of the ellipse is determined by the sum of these two distances. In JavaFX, an ellipse is represented by a class named **Ellipse.** This class belongs to the package **javafx.scene.shape**. This class has 4 properties of the double datatype namely – • **centerX:** The x coordinate of the center of the ellipse in pixels. • **centerY:** The y coordinate of the center of the ellipse in pixels. • **radiusX:** The width of the ellipse pixels. • **radiusY:** The height of the ellipse pixels.
6	**Polygon:** A closed shape formed by a number of coplanar line segments connected end to end. In JavaFX, a polygon is represented by a class named **Polygon**. This class belongs to the package **javafx.scene.shape**.
7	**Polyline:** A polyline is same a polygon except that a polyline is not closed in the end. Or, continuous line composed of one or more line segments. In JavaFX, a Polyline is represented by a class named **Polygon**. This class belongs to the package **javafx.scene.shape**.
8	**Cubic curve:** A cubic curve is a Bezier parametric curve in the XY plane is a curve of degree 3.

	In JavaFX, a Cubic Curve is represented by a class named **CubicCurve.** This class belongs to the package **javafx.scene.shape**. This class has 8 properties of the double datatype namely – • **startX:** The x coordinate of the starting point of the curve. • **startY:** The y coordinate of the starting point of the curve. • **controlX1:** The x coordinate of the first control point of the curve. • **controlY1:** The y coordinate of the first control point of the curve. • **controlX2:** The x coordinate of the second control point of the curve. • **controlY2:** The y coordinate of the second control point of the curve. • **endX:** The x coordinate of the end point of the curve. • **endY:** The y coordinate of the end point of the curve.
9	**Quad Curve:** A quadratic curve is a Bezier parametric curve in the XY plane is a curve of degree 2. In JavaFX, a QuadCurve is represented by a class named QuadCurve**.** This class belongs to the package **javafx.scene.shape**. • **startX:** The x coordinate of the starting point of the curve. • **startY:** The y coordinate of the starting point of the curve. • **controlX:** The x coordinate of the control point of the curve. • **controlY:** The y coordinate of the control point of the curve. • **endX:** The x coordinate of the end point of the curve • **endY:** The y coordinate of the end point of the curve.
10	**Arc:** An arc is part of a curve. In JavaFX, an arc is represented by a class named **Arc.** This class belongs to the package – **javafx.scene.shape**. This class has 4 properties of the double datatype namely – • **centerX:** The x coordinate of the center of the arc. • **centerY:** The y coordinate of the center of the arc. • **radiusX:** The width of the full ellipse of which the current arc is a part of. • **radiusY:** The height of the full ellipse of which the current arc is a part of. • **startAngle:** The starting angle of the arc in degrees. • **length:** The angular extent of the arc in degrees.

11	**SVG shape:** In JavaFX, we can construct images by parsing SVG paths. Such shapes are represented by the class named **SVGPath.** This class belongs to the package **javafx.scene.shape**. This class has a property named **content** of String datatype. This represents the SVG Path encoded string, from which the image should be drawn.

2D Shapes – Line

In general, a line is a geometrical structure which joins two points on an XY plane.

In JavaFX, a line is represented by a class named **Line.** This class belongs to the package **javafx.scene.shape.**

By instantiating this class, you can create a line node in JavaFX.

This class has 4 properties of the double datatype namely –

- **startX:** The x coordinate of the start point of the line.
- **startY:** The y coordinate of the start point of the line.
- **endX:** The x coordinate of the end point of the line.
- **endY:** The y coordinate of the end point of the line.

To draw a line, you need to pass values to these properties, either by passing them to the constructor of this class, in the same order, at the time of instantiation, as follows –

```
Line line = new Line(startX, startY, endX, endY);
```

Or, by using their respective setter methods as follows –

```
setStartX(value);
setStartY(value);
setEndX(value);
setEndY(value);
```

Follow the steps given below to Draw a Line in JavaFX.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application** and implement the **start()** method of this class as follows.

```
public class ClassName extends Application {


    @Override
    public void start(Stage primaryStage) throws Exception {


    }


}
```

Step 2: Creating a line

You can create a line in JavaFX by instantiating the class named **Line** which belongs to a package **javafx.scene.shape**, instantiate this class as follows.

```
//Creating a line object
Line line = new Line();
```

Step 3: Setting Properties to the Line

Specify the coordinates to draw the line on an XY plane by setting the properties startX, startY, endX and endY, using their respective setter methods as shown in the following code block.

```
line.setStartX(100.0);
line.setStartY(150.0);
line.setEndX(500.0);
line.setEndY(150.0);
```

Step 4: Creating a Group Object

Create a group object by instantiating the class named **Group**, which belongs to the package **javafx.scene**.

Pass the Line (node) object, created in the previous step, as a parameter to the constructor of the Group class, in order to add it to the group as follows –

```
Group root = new Group(line);
```

Step 5: Creating a Scene Object

Create a Scene by instantiating the class named **Scene** which belongs to the package **javafx.scene.** To this class pass the Group object (**root**), created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 6: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object which is passed to the start method of the scene class, as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 7: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene Object prepared in the previous steps using this method as follows.

```
primaryStage.setScene(scene);
```

Step 8: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 9: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows.

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

Example

Following is the program which generates a straight line using JavaFX. Save this code in a file with the name **DrawingLine.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.shape.Line;
import javafx.stage.Stage;

public class DrawingLine extends Application{

   @Override
   public void start(Stage stage) {

      //Creating a line object
      Line line = new Line();

      //Setting the properties to a line
      line.setStartX(100.0);
      line.setStartY(150.0);
      line.setEndX(500.0);
      line.setEndY(150.0);

      //Creating a Group
      Group root = new Group(line);

      //Creating a Scene
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the scene
      stage.setTitle("Sample application");

      //Adding the scene to the stage
      stage.setScene(scene);

      //Displaying the contents of a scene
      stage.show();

    }
```

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

}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac DrawingLine.java
java DrawingLine
```

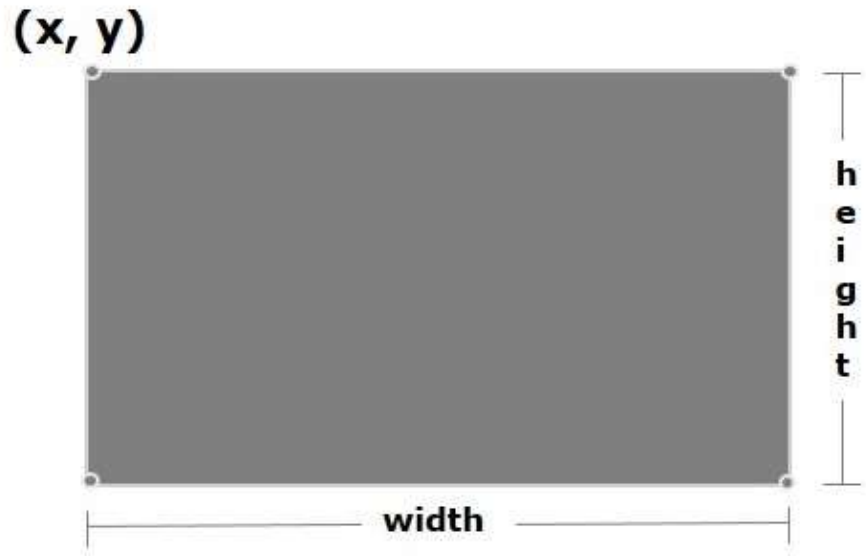
On executing, the above program generates a JavaFX window displaying a straight line as shown below.

2D Shapes – Rectangle

In general, a rectangle is a four-sided polygon that has two pairs of parallel and concurrent sides with all interior angles as right angles.

It is described by two parameters namely —

- **height**: The vertical length of the rectangle is known as height.
- **width:** The horizontal length of the rectangle is known as width.



In JavaFX, a rectangle is represented by a class named **Rectangle.** This class belongs to the package **javafx.scene.shape**.

By instantiating this class, you can create a Rectangle node in JavaFX.

This class has 4 properties of the double datatype namely –

- **x:** The x coordinate of the start point (upper left) of the rectangle.
- **y:** The y coordinate of the start point (upper left) of the rectangle.
- **width:** The width of the rectangle.
- **height:** The height of the rectangle.

To draw a rectangle, you need to pass values to these properties, either by passing them to the constructor of this class, in the same order, at the time of instantiation, as shown below:

```
Rectangle rectangle = new Rectangle(x, y, width, height);
```

Or, by using their respective setter methods as shown in the following code block –

```
setX(value);
setY(value);
setWidth(value);
setHeight(value);
```

You need to follow the steps given below to draw a rectangle in JavaFX.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application** and implement the **start()** method of this class as shown below.

```
public class ClassName extends Application {


    @Override
    public void start(Stage primaryStage) throws Exception {


    }


}
```

Step 2: Creating a Rectangle

You can create a rectangle in JavaFX by instantiating the class named **Rectangle** which belongs to a package **javafx.scene.shape**, instantiate this class as follows.

```
//Creating a rectangle object
Rectangle rectangle = new Rectangle();
```

Step 3: Setting Properties to the Rectangle

Specify the x, y coordinates of the starting point (upper left), height and the width of the rectangle, that is needed to be drawn. You can do this by setting the properties x, y, height and width, using their respective setter methods as shown in the following code block.

```
line.setStartX(100.0);
line.setStartY(150.0);
line.setEndX(500.0);
line.setEndY(150.0);
```

Step 4: Creating a Group Object

In the **start()** method, create a group object by instantiating the class named **Group**, which belongs to the package **javafx.scene**.

Pass the Rectangle (node) object, created in the previous step, as a parameter to the constructor of the Group class, in order to add it to the group as follows –

```
Group root = new Group(rectangle);
```

Step 5: Creating a Scene Object

Create a Scene by instantiating the class named **Scene** that belongs to the package **javafx.scene.** To this class, pass the Group object (**root**), created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 6: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object, which is passed to the start method of the scene class, as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 7: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using the method shown below.

```
primaryStage.setScene(scene);
```

Step 8: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 9: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows.

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

Example

Following is the program which generates a rectangle JavaFX. Save this code in a file with the name **RectangleExample.java.**

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

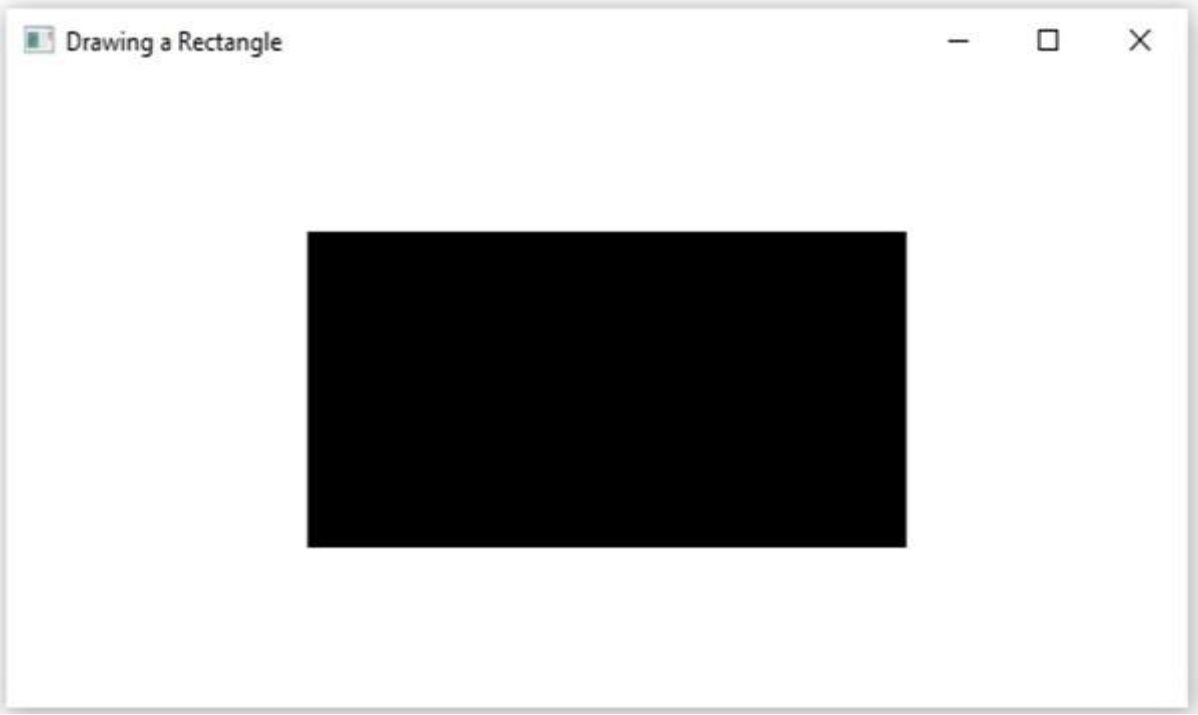
```
public class RectangleExample extends Application {
   @Override
   public void start(Stage stage) {

      //Drawing a Rectangle
      Rectangle rectangle = new Rectangle();

      //Setting the properties of the rectangle
      rectangle.setX(150.0f);
      rectangle.setY(75.0f);
      rectangle.setWidth(300.0f);
      rectangle.setHeight(150.0f);

      //Creating a Group object
      Group root = new Group(rectangle);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Drawing a Rectangle");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }

   public static void main(String args[]){

      launch(args);

   }
```

```
}
```

Compile and execute the saved java file from the command prompt using the following commands.

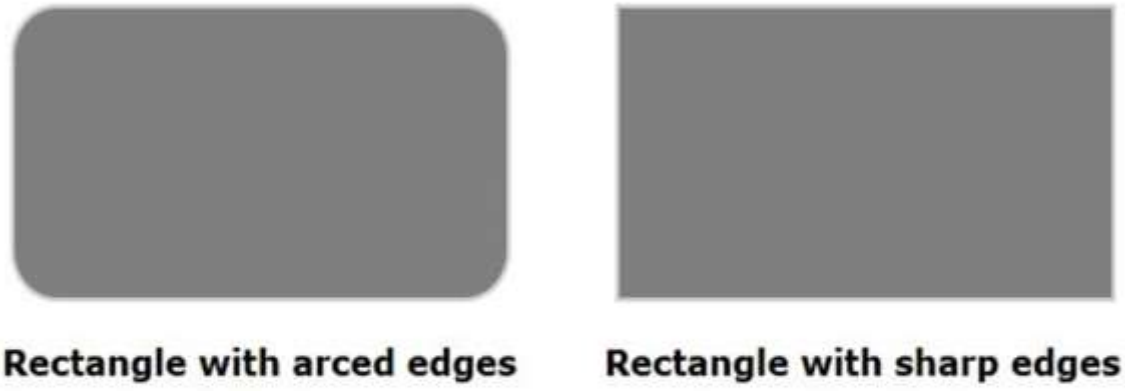
```
javac RectngleExample.java
java RectangleExample
```

On executing, the above program generates a JavaFX window displaying a rectangle as shown in the following screenshot.



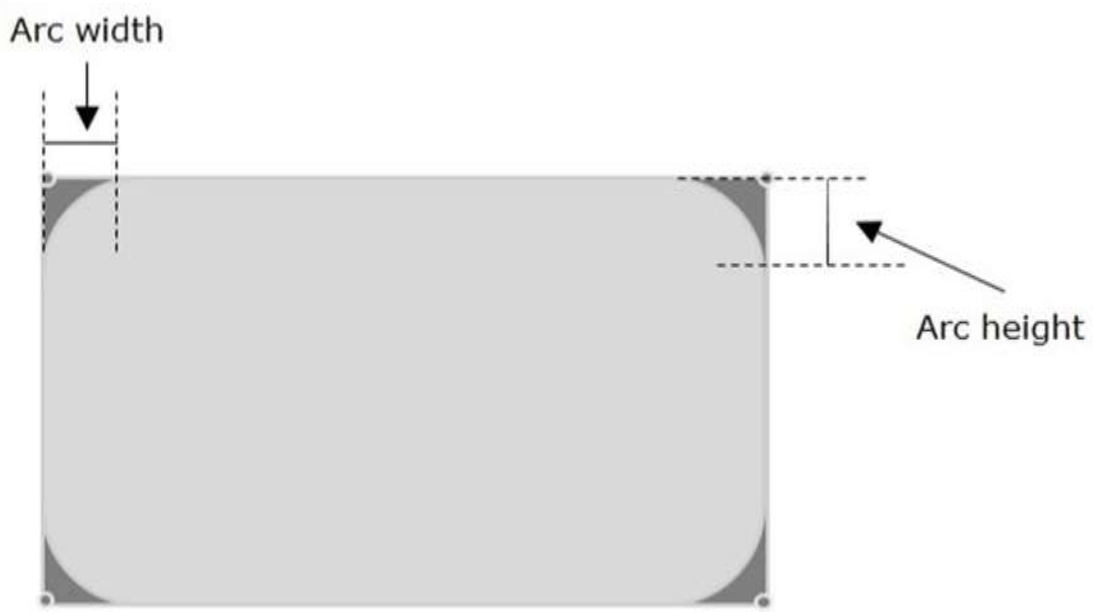
Rounded Rectangle

In JavaFX, you can draw a rectangle either with sharp edges or with arched edges as shown in the following diagram.



The one with arched edges is known as a rounded rectangle and it has two additional properties namely –

- **arcHeight:** The vertical diameter of the arc, at the corners of a rounded rectangle.
- **arcWidth:** The horizontal diameter of the arc at the corners of a rounded rectangle.



By default, JavaFX creates a rectangle with sharp edges unless you set the height and width of the arc to +ve values (0<) using their respective setter methods **setArcHeight()** and **setArcWidth().**

Example

Following is a program which generates a rounded rectangle using JavaFX. Save this code in a file with the name **RoundedRectangle.java.**

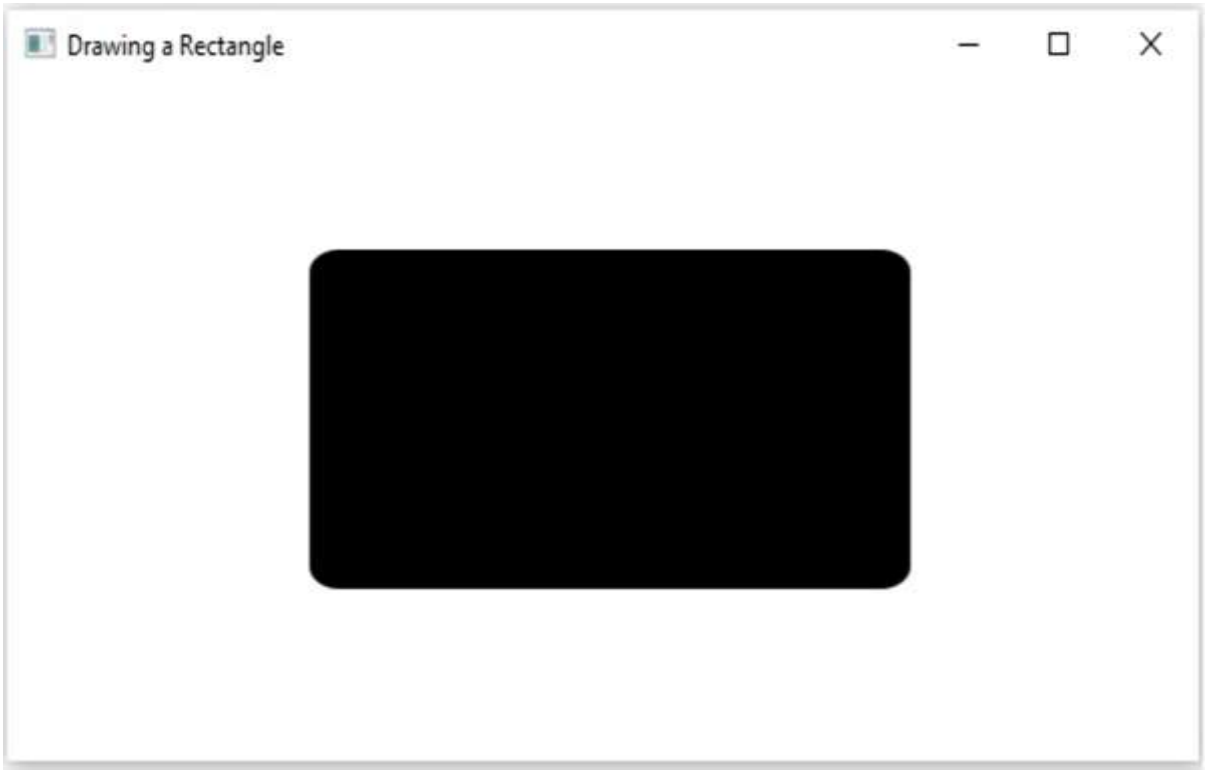
```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.shape.Rectangle;

public class RoundedRectangle extends Application {
   @Override
   public void start(Stage stage) {
      //Drawing a Rectangle
      Rectangle rectangle = new Rectangle();

      //Setting the properties of the rectangle
      rectangle.setX(150.0f);
      rectangle.setY(75.0f);
      rectangle.setWidth(300.0f);
      rectangle.setHeight(150.0f);

      // Setting the height and width of the arc
      rectangle.setArcWidth(30.0);
      rectangle.setArcHeight(20.0);

      // Creating a Group object
      Group root = new Group(rectangle);

      // Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      // Setting title to the Stage
      stage.setTitle("Drawing a Rectangle");

      // Adding scene to the satge
      stage.setScene(scene);
```

```
      // Displaying the contents of the stage
      stage.show();
   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac RoundedRectangle.java
java RoundedRectangle
```

On executing, the above program generates a JavaFX window displaying a rounded rectangle as shown below.

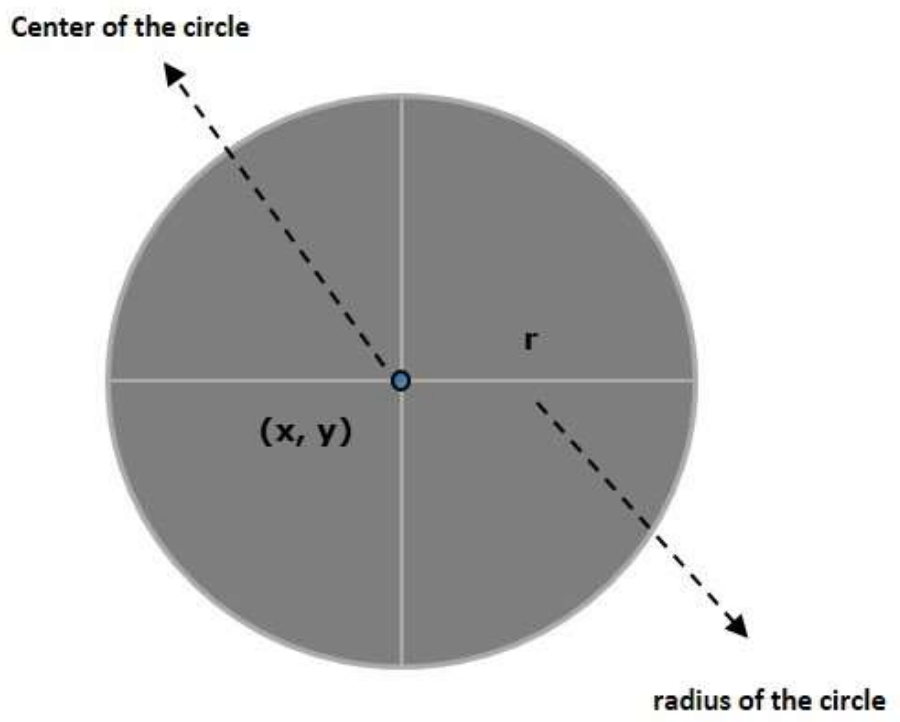


2D Shapes – Circle

A circle is the locus of all points at a fixed distance (radius of circle) from a fixed point (the centre of circle). In other words, a circle is a line forming a closed loop, every point on which is a fixed distance from a centre point.

A circle is defined by two parameters namely –

- **Centre:** It is a point inside the circle. All points on the circle are equidistant (same distance) from the centre point.
- **Radius**: The radius is the distance from the centre to any point on the circle. It is half the diameter.



In JavaFX, a circle is represented by a class named **Circle.** This class belongs to the package **javafx.scene.shape**.

By instantiating this class, you can create a Circle node in JavaFX.

This class has 3 properties of the double datatype namely –

- **centerX:** The x coordinate of the center of a circle.
- **centerY:** The y coordinate of the center of a circle.
- **radius:** The radius of the circle in pixels.

To draw a circle, you need to pass values to these properties, either by passing them to the constructor of this class, in the same order, at the time of instantiation, as follows –

```
Circle circle = new Circle(centerx, centery, radius);
```

Or, by using their respective setter methods as follows –

```
setCenterX(value);
setCenterY(value);
setRadius(value);
```

Follow the steps given below to draw a Circle in JavaFX.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application** and implement the **start()** method of this class as follows.

```
public class ClassName extends Application {


    @Override
    public void start(Stage primaryStage) throws Exception {


    }


}
```

Step 2: Creating a Circle

You can create a circle in JavaFX by instantiating the class named **Circle** which belongs to a package **javafx.scene.shape**, instantiate this class as follows.

```
//Creating a circle object
Circle circle = new Circle();
```

Step 3: Setting Properties to the Circle

Specify the x, y coordinates of the center of the circle and the radius of the circle by setting the properties X, Y, and radius using their respective setter methods as shown in the following code block.

```
circle.setCenterX(300.0f);
circle.setCenterY(135.0f);
circle.setRadius(100.0f);
```

Step 4: Creating a Group Object

In the **start()** method, create a group object by instantiating the class named **Group**, which belongs to the package **javafx.scene**.

Pass the circle (node) object, created in the previous step, as a parameter to the constructor of the Group class, in order to add it to the group as follows:

```
Group root = new Group(circle);
```

Step 5: Creating a Scene Object

Create a Scene by instantiating the class named **Scene** which belongs to the package **javafx.scene**. To this class, pass the Group object (**root**), created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 6: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object which is passed to the start method of the scene class, as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 7: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using this method as follows.

```
primaryStage.setScene(scene);
```

Step 8: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 9: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows.

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

Example

Following is a program which generates a circle using JavaFX. Save this code in a file with the name **CircleExample.java.**

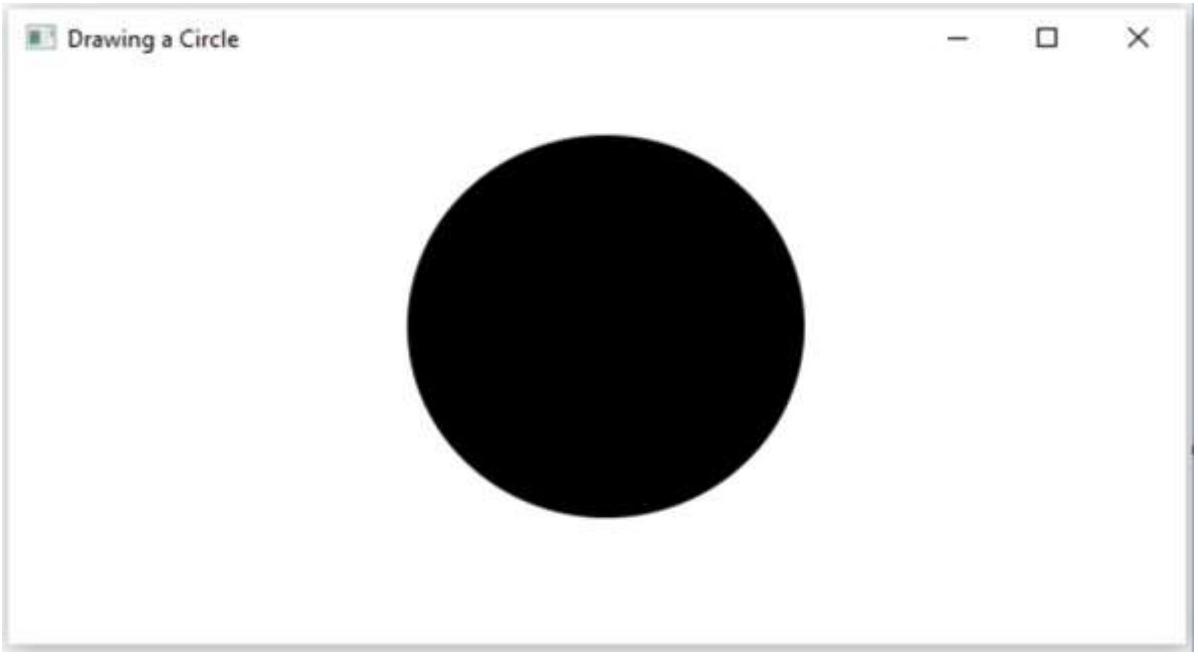
```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.shape.Circle;

public class CircleExample extends Application {
   @Override
   public void start(Stage stage) {

      //Drawing a Circle
      Circle circle = new Circle();

      //Setting the properties of the circle
      circle.setCenterX(300.0f);
      circle.setCenterY(135.0f);
      circle.setRadius(100.0f);

      //Creating a Group object
      Group root = new Group(circle);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Drawing a Circle");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
```

```
    }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac CircleExample.java
java CircleExample
```

On executing, the above program generates a javaFx window displaying a circle as shown below.



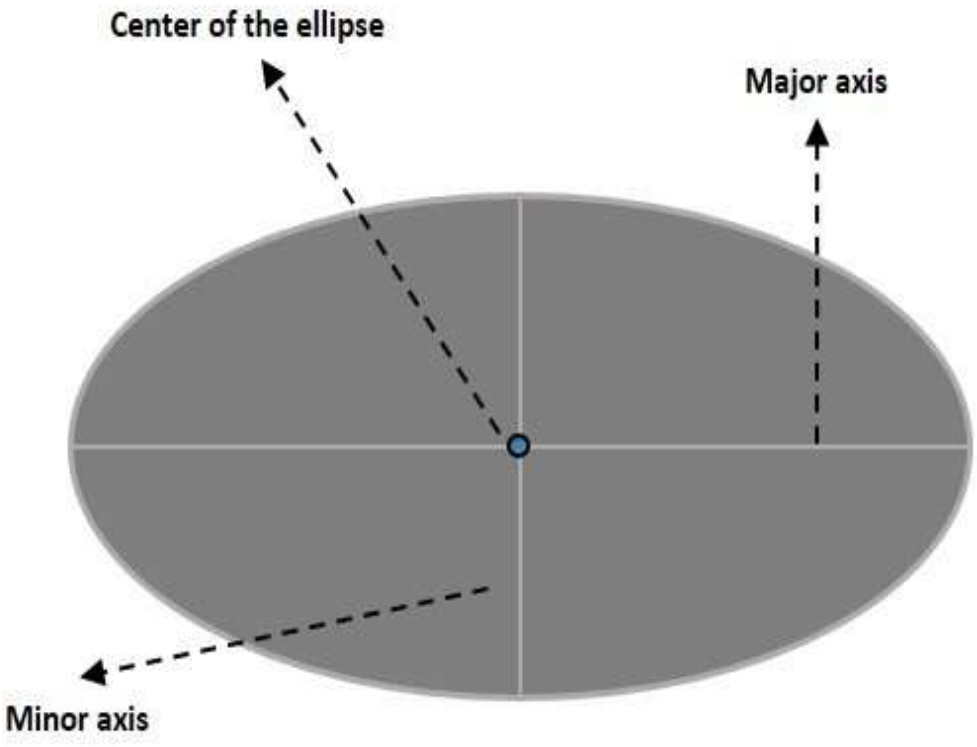
2D Shapes – Ellipse

An Ellipse is defined by two points, each called a focus. If any point on the Ellipse is taken, the sum of the distances to the focus points is constant. The size of the Ellipse is determined by the sum of these two distances. The sum of these distances is equal to the length of the major axis (the longest diameter of the ellipse). A circle is, in fact, a special case of an Ellipse.

An Ellipse has three properties which are –

- **Centre:** A point inside the Ellipse which is the midpoint of the line segment linking the two foci. The intersection of the major and minor axes.

- **Major axis:** The longest diameter of an ellipse.
- **Minor axis:** The shortest diameter of an ellipse.



In JavaFX, an Ellipse is represented by a class named **Ellipse.** This class belongs to the package **javafx.scene.shape**.

By instantiating this class, you can create an Ellipse node in JavaFX.

This class has 4 properties of the double datatype namely –

- **centerX:** The x coordinate of the center of the Ellipse in pixels.
- **centerY:** The y coordinate of the center of the Ellipse in pixels.
- **radiusX:** The width of the Ellipse pixels.
- **radiusY:** The height of the Ellipse pixels.

To draw an ellipse, you need to pass values to these properties, either by passing them to the constructor of this class, in the same order, at the time of instantiation, as shown below –

```
Circle circle = new Circle(centerX, centerY, radiusX, radiusY);
```

Or, by using their respective setter methods as follows –

```
setCenterX(value);
setCenterY(value);
```

```
setRadiusX(value);
setRadiusY(value);
```

Follow the steps given below to draw an Ellipse in JavaFX.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application** and implement the **start()** method of this class as shown below.

```
public class ClassName extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
    }
}
```

Step 2: Creating an Ellipse

You can create an Ellipse in JavaFX by instantiating the class named ***Line*** which belongs to a package **javafx.scene.shape.** You can instantiate this class as follows.

```
//Creating an Ellipse object
Ellipse ellipse = new Ellipse();
```

Step 3: Setting Properties to the Ellipse

Specify the x, y coordinates of the center of the Ellipse → the width of the Ellipse along x axis and y axis (major and minor axises), of the circle by setting the properties X, Y, RadiusX and RadiusY.

This can be done by using their respective setter methods as shown in the following code block.

```
ellipse.setCenterX(300.0f);
ellipse.setCenterY(150.0f);
ellipse.setRadiusX(150.0f);
ellipse.setRadiusY(75.0f);
```

Step 4: Creating a Group Object

In the **start()** method, create a group object by instantiating the class named **Group**, which belongs to the package **javafx.scene**.

Pass the Ellipse (node) object created in the previous step as a parameter to the constructor of the Group class. This should be done in order to add it to the group as shown in the following code block –

```
Group root = new Group(ellipse);
```

Step 5: Creating a Scene Object

Create a Scene by instantiating the class named **Scene** which belongs to the package **javafx.scene.** To this class pass the Group object (**root**) created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 6: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object which is passed to the start method of the scene class as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 7: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous step using this method as follows.

```
primaryStage.setScene(scene);
```

Step 8: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 9: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows.

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

Example

Following is a program which generates an Ellipse using JavaFX. Save this code in a file with the name **EllipseExample.java**.

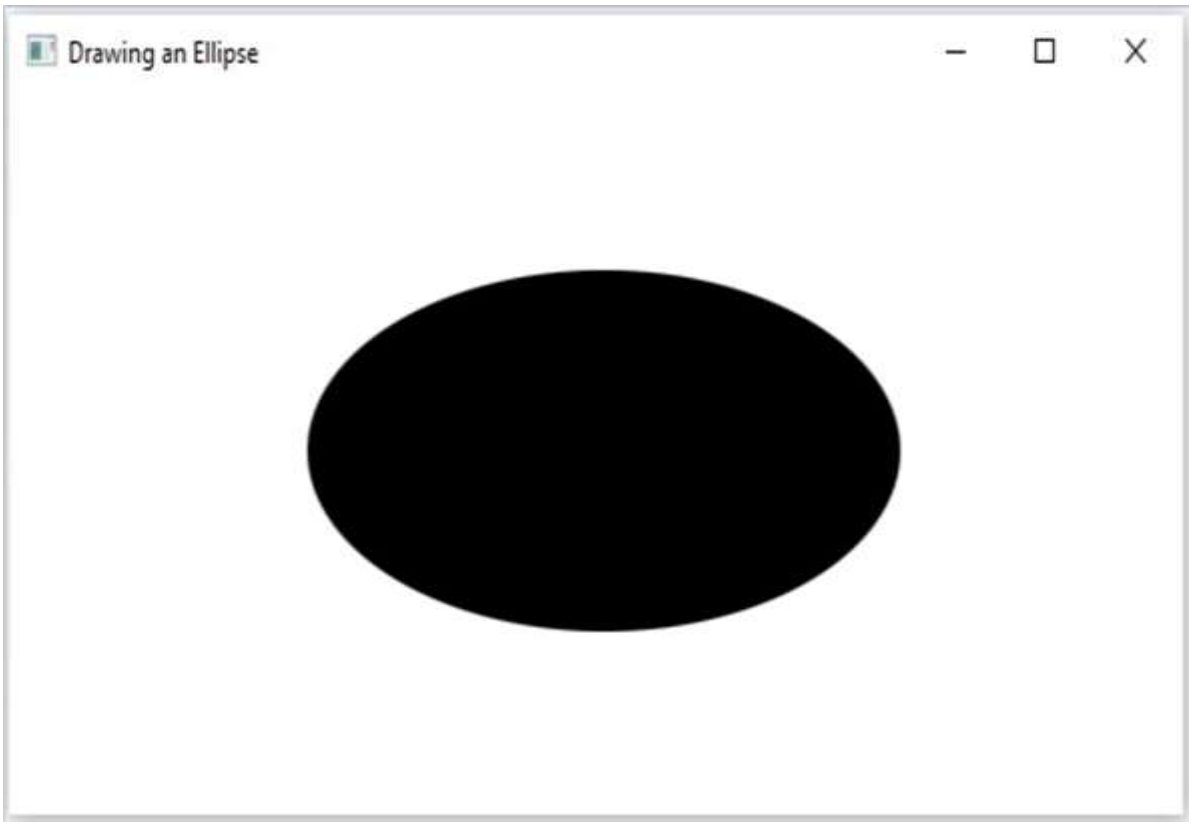
```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.shape.Ellipse;

public class EllipseExample extends Application {

   @Override
   public void start(Stage stage) {

      //Drawing an ellipse
      Ellipse ellipse = new Ellipse();

      //Setting the properties of the ellipse
      ellipse.setCenterX(300.0f);
      ellipse.setCenterY(150.0f);
      ellipse.setRadiusX(150.0f);
      ellipse.setRadiusY(75.0f);

      //Creating a Group object
      Group root = new Group(ellipse);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);
      //Setting title to the Stage
      stage.setTitle("Drawing an Ellipse");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
```

```
    }

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

Compile and execute the saved Java file from the command prompt using the following commands.

```
javac EllipseExample.java
java EllipseExample
```

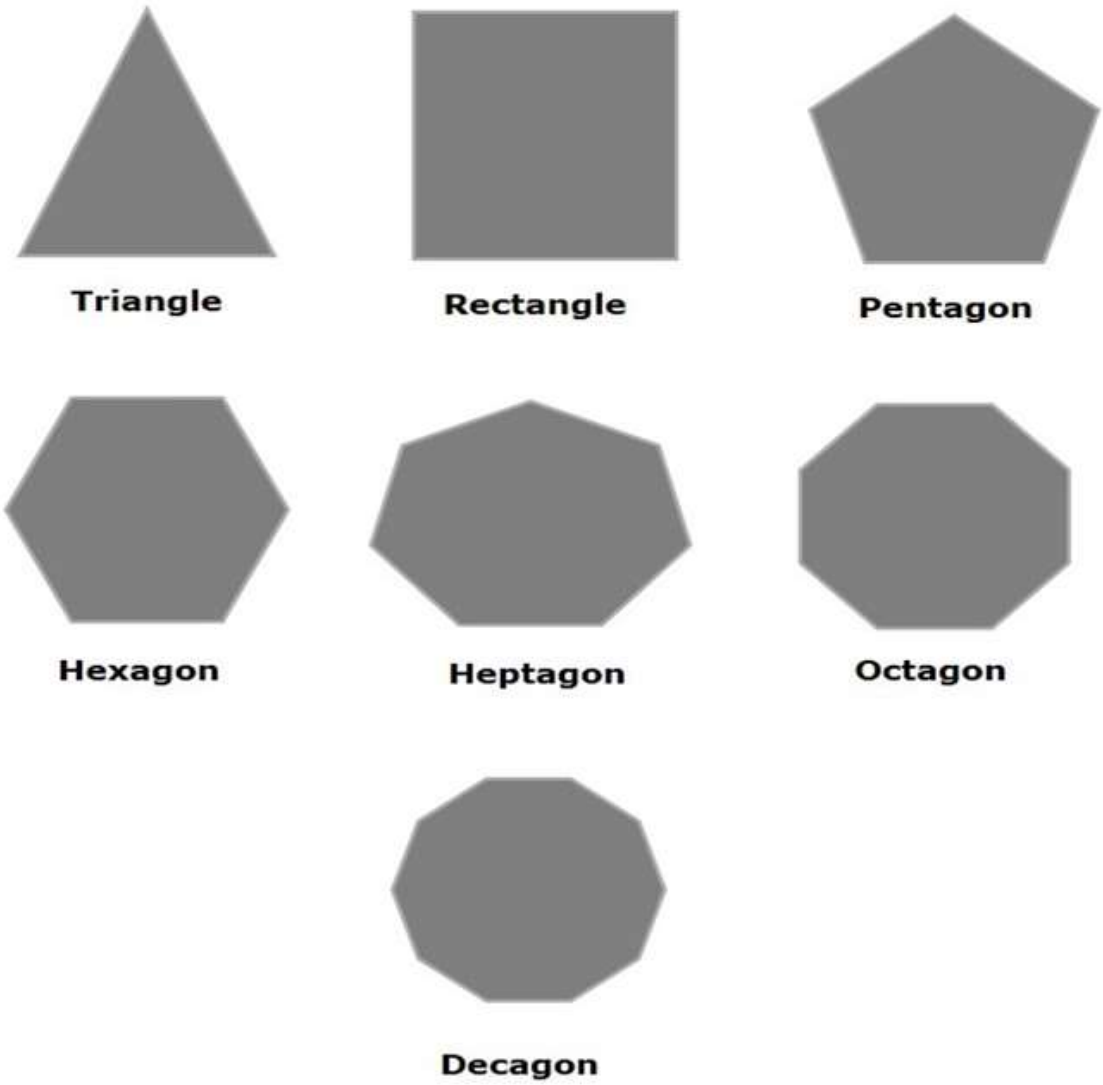
On executing, the above program generates a JavaFX window displaying an ellipse as shown below.



2D Shapes – Polygon

A closed shape formed by a number of coplanar line segments connected end to end.

A polygon is described by two parameters, namely, the length of its sides and the measures of its interior angles.



In JavaFX, a polygon is represented by a class named **Polygon**. This class belongs to the package **javafx.scene.shape**.

By instantiating this class, you can create a polygon node in JavaFX. You need to pass the x, y coordinates of the points by which the polygon should be defined in the form of a double array.

You can pass the double array as a parameter of the constructor of this class as shown below –

```
Polygon polygon = new Plygon(doubleArray);
```

Or, by using the **getPoints()** method as follows –

```
polygon.getPoints().addAll(new Double[]{ List of XY coordinates separated by
commas });
```

To draw a polygon in JavaFX, follow the steps given below.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application** and implement the **start()** method of this class as follows.

```
public class ClassName extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
    }
}
```

Step 2: Creating a Polygon

You can create a polygon in JavaFX by instantiating the class named **Polygon** which belongs to a package **javafx.scene.shape.** You can instantiate this class as follows.

```
//Creating an object of the class Polygon
Polygon hexagon = new Polygon();
```

Step 3: Setting Properties to the Polygon

Specify a double array holding the XY coordinates of the points of the required polygon (hexagon in this example) separated by commas, using the **getPoints()** method of the **Polygon** class, as follows.

```
//Adding coordinates to the hexagon
hexagon.getPoints().addAll(new Double[]{
   200.0, 50.0,
   400.0, 50.0,
   450.0, 150.0,
   400.0, 250.0,
   200.0, 250.0,
   150.0, 150.0,
   });
```

Step 4: Creating a Group Object

In the **start()** method, create a group object by instantiating the class named **Group**, which belongs to the package **javafx.scene**.

Pass the polygon node (hexagon) object, created in the previous step, as a parameter to the constructor of the Group class, in order to add it to the group as follows –

```
Polygon root = new Polygon(hexagon);
```

Step 5: Creating a Scene Object

Create a Scene by instantiating the class named **Scene** which belongs to the package **javafx.scene.** To this class pass the Group object (**root**), created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 6: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is the Stage object which is passed to the start method of the scene class as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 7: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous step using the method as shown below.

```
primaryStage.setScene(scene);
```

Step 8: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 9: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows.

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

Example

Following is a program which generates a Polygon (hexagon) using JavaFX. Save this code in a file with the name **PolygonExample.java**.

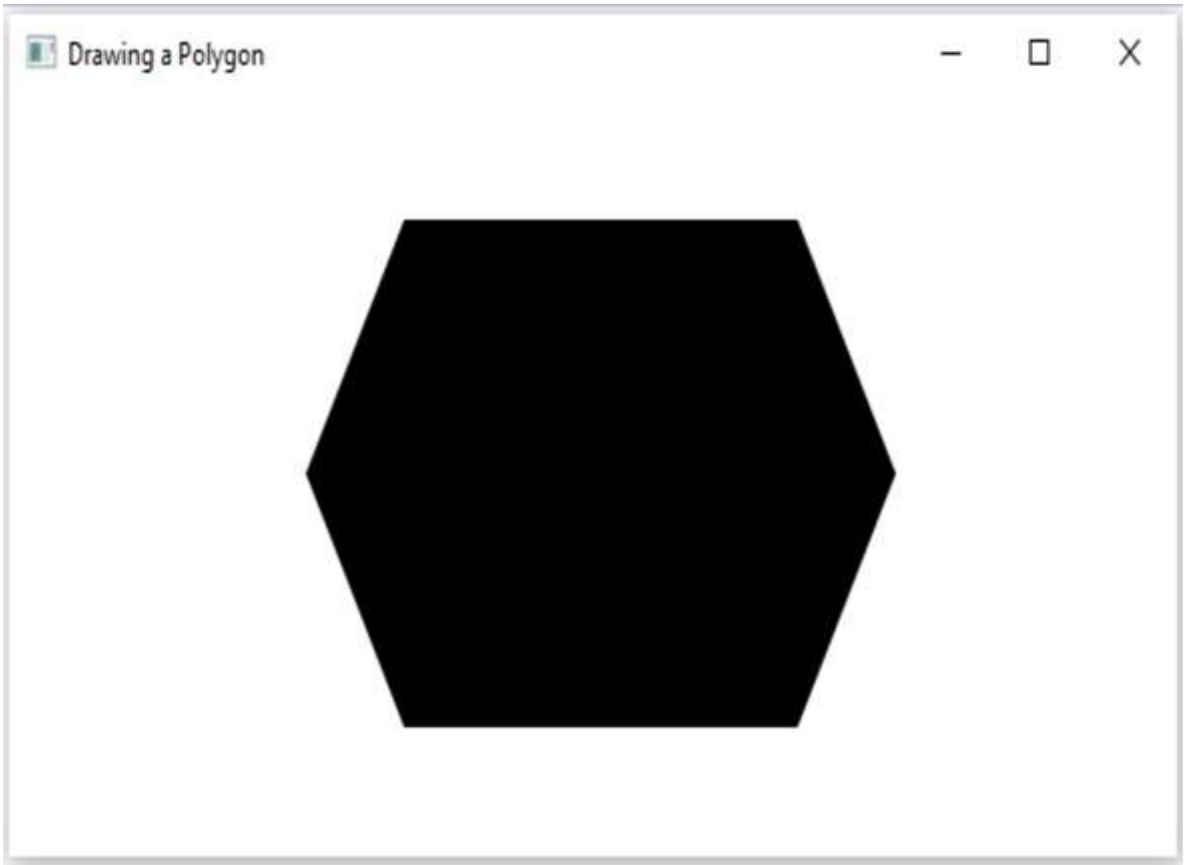
```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.shape.Polygon;
import javafx.stage.Stage;

public class PolygonExample extends Application {
   @Override
   public void start(Stage stage) {

      //Creating a Polygon
      Polygon polygon = new Polygon();

      //Adding coordinates to the polygon
      polygon.getPoints().addAll(new Double[]{
         300.0, 50.0,
         450.0, 150.0,
         300.0, 250.0,
         150.0, 150.0,
          });

      //Creating a Group object
      Group root = new Group(polygon);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Drawing a Polygon");

      //Adding scene to the satge
      stage.setScene(scene);
```

```
      //Displaying the contents of the stage
      stage.show();
   }
   public static void main(String args[]){
      launch(args);
   }
}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac PolygonExample.java
java PolygonExample
```

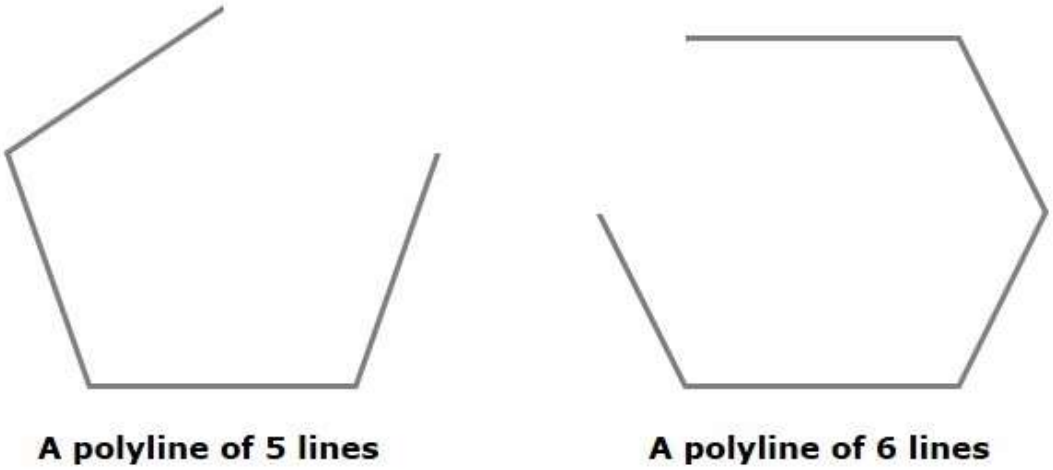
On executing, the above program generates a JavaFX window displaying a polygon as shown below.



2D Shapes – Polyline

A Polyline is same as a polygon except that a polyline is not closed in the end. Or, continuous line composed of one or more line segments.

In short, we can say a polygon is an open figure formed by coplanar line segments.



A polyline of 5 lines

A polyline of 6 lines

In JavaFX, a Polyline is represented by a class named **Polygon**. This class belongs to the package **javafx.scene.shape**.

By instantiating this class, you can create polyline node in JavaFX. You need to pass the x, y coordinates of the points by which the polyline should be defined in the form of a double array.

You can pass the double array as a parameter of the constructor of this class as shown below –

```
Polyline polyline = new Polyline(doubleArray);
```

Or, by using the **getPoints()** method as follows –

```
polyline.getPoints().addAll(new Double[]{List of XY coordinates separated by
commas });
```

To Draw a Polyline in JavaFX, follow the steps given below.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application** and implement the **start()** method of this class as follows.

```
public class ClassName extends Application {

    @Override

    public void start(Stage primaryStage) throws Exception {
```

```
    }
}
```

Step 2: Creating a Polyline

You can create a line in JavaFX by instantiating the class named **Line** which belongs to a package **javafx.scene.shape.** You can instantiate this class as follows.

```
//Creating an object of the class Polyline
Polyline polyline = new Polyline();
```

Step 3: Setting Properties to the Polyline

Specify a double array holding the XY coordinates of the points of the required polyline (hexagon in this example) separated by commas. You can do this by using the **getPoints()** method of the **Polyline** class as shown in the following code block.

```
//Adding coordinates to the hexagon
polyline.getPoints().addAll(new Double[]{
   200.0, 50.0,
   400.0, 50.0,
   450.0, 150.0,
   400.0, 250.0,
   200.0, 250.0,
   150.0, 150.0,
   });
```

Step 4: Creating a Group Object

In the **start()** method create a group object by instantiating the class named **Group**, which belongs to the package **javafx.scene**.

Pass the **Polyline** (node) object, created in the previous step, as a parameter to the constructor of the Group class, in order to add it to the group as follows –

```
Group root = new Group(polyline);
```

Step 5: Creating a Scene Object

Create a Scene by instantiating the class named **Scene** which belongs to the package **javafx.scene.** To this class pass the Group object (**root**) that was created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 6: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object which is passed to the start method of the scene class as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 7: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using the following method.

```
primaryStage.setScene(scene);
```

Step 8: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 9: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows.

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

Example

Following is a program which generates a polyline using JavaFX. Save this code in a file with the name **PolylineExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.shape.Polyline;
```

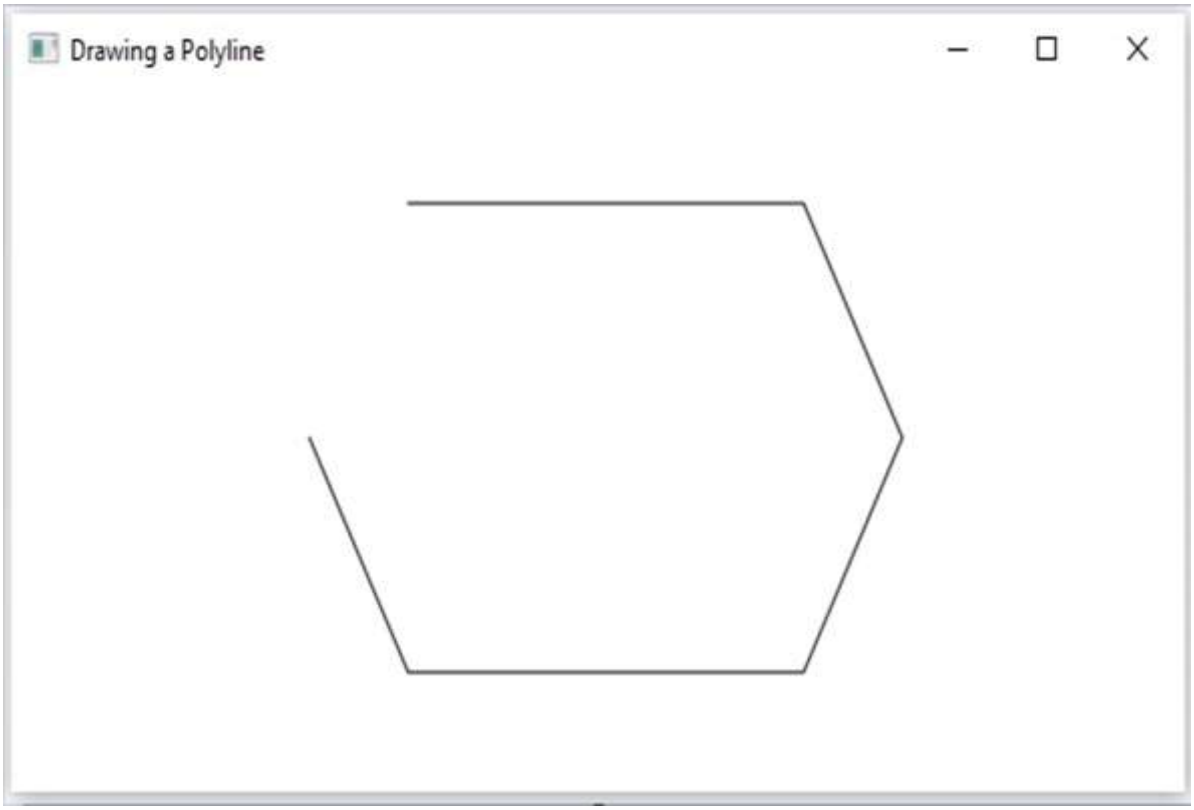
```
public class PolylineExample extends Application {

   @Override
   public void start(Stage stage) {

      //Creating a polyline
      Polyline polyline = new Polyline();

      //Adding coordinates to the polygon
      polyline.getPoints().addAll(new Double[]{
         200.0, 50.0,
         400.0, 50.0,
         450.0, 150.0,
         400.0, 250.0,
         200.0, 250.0,
         150.0, 150.0,
        });

      //Creating a Group object
      Group root = new Group(polyline);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Drawing a Polyline");

      //Adding scene to the stage
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }
```

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac PolylineExample.java
java PolylineExample
```

On executing, the above program generates a JavaFX window displaying a polyline as shown below.



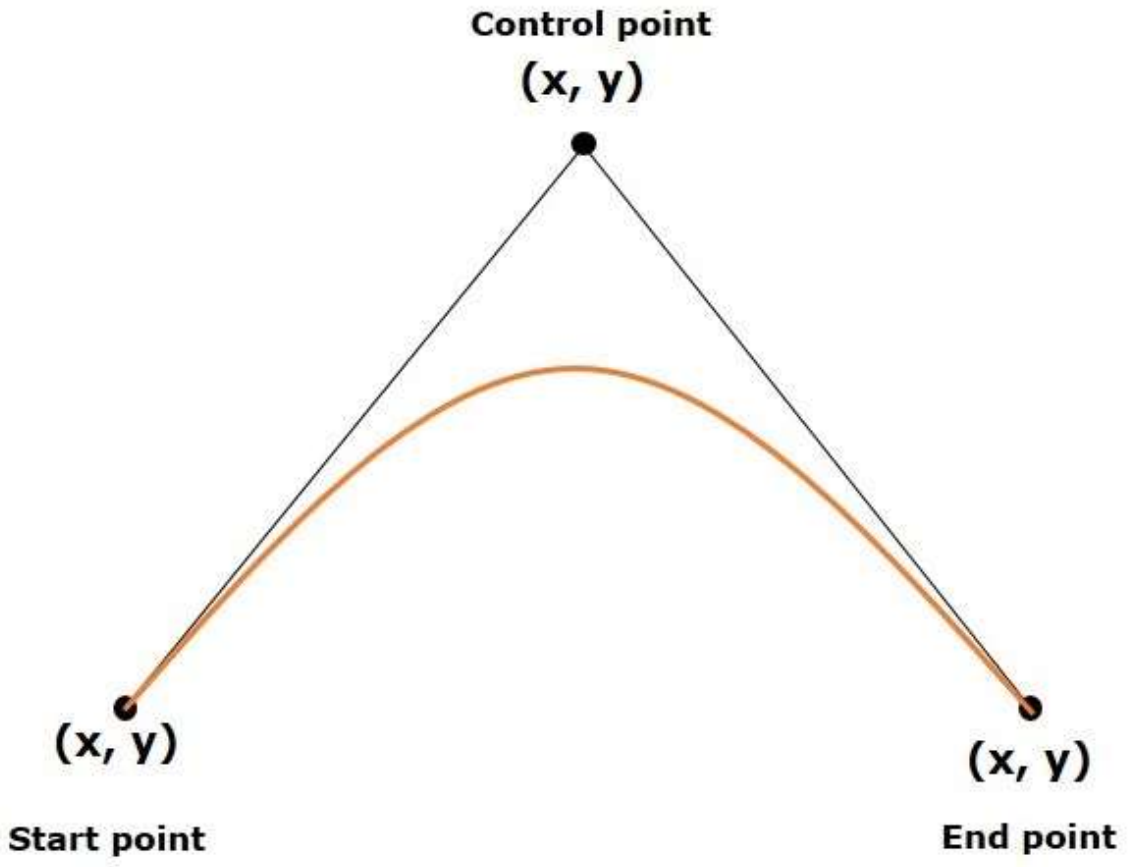
2D Shapes – QuadCurve

Mathematically a quadratic curve is one that is described by a quadratic function like –

$y = ax^2 + bx + c$.

In computer graphics Bezier curves are used. These are parametric curves which appear reasonably smooth at all scales. These Bezier curves are drawn based on points on an XY plane.

A quadratic curve is a Bezier parametric curve in the XY plane which is a curve of degree 2. It is drawn using three points: **start point**, **end point** and **control point** as shown in the following diagram.



In JavaFX, a QuadCurve is represented by a class named **QuadCurve.** This class belongs to the package **javafx.scene.shape**.

By instantiating this class, you can create a QuadCurve node in JavaFX.

This class has 6 properties of the double datatype namely –

- **startX:** The x coordinate of the starting point of the curve.
- **startY:** The y coordinate of the starting point of the curve.
- **controlX:** The x coordinate of the control point of the curve.
- **controlY:** The y coordinate of the control point of the curve.
- **endX:** The x coordinate of the end point of the curve
- **endY:** The y coordinate of the end point of the curve.

To draw a QuadCurve, you need to pass values to these properties. This can be done either by passing them to the constructor of this class, in the same order, at the time of instantiation, as follows –

```
QuadCurve quadcurve = new QuadCurve(startX, startY, controlX, controlY, endX,
endY);
```

Or, by using their respective setter methods as follows –

```
setStartX(value);
setStartY(value);
setControlX(value);
setControlY(value);
setEndX(value);
setEndY(value);
```

To Draw a Bezier Quadrilateral Curve in JavaFX, follow the steps given below.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application**. Then you can implement the **start()** method of this class as follows.

```
public class ClassName extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
    }
}
```

Step 2: Creating a QuadCurve

You can create a QuadCurve in JavaFX by instantiating the class named **QuadCurve** which belongs to a package **javafx.scene.shape**. You can then instantiate this class as shown in the following code block.

```
//Creating an object of the class QuadCurve
QuadCurve quadcurve = new QuadCurve();
```

Step 3: Setting Properties to the QuadCurve

Specify the x, y coordinates of the three points: **start point**, **end point** and **control points**, of the required curve, using their respective setter methods as shown in the following code block.

```
//Adding properties to the Quad Curve
quadCurve.setStartX(100.0);
quadCurve.setStartY(220.0f);
quadCurve.setEndX(500.0f);
quadCurve.setEndY(220.0f);
```

```
quadCurve.setControlX(250.0f);
quadCurve.setControlY(0.0f);
```

Step 4: Creating a Group Object

In the **start()** method, create a group object by instantiating the class named **Group**, which belongs to the package **javafx.scene**.

Pass the **QuadCurve** (node) object created in the previous step as a parameter to the constructor of the Group class, in order to add it to the group as follows –

```
Group root = new Group(quadcurve);
```

Step 5: Creating a Scene Object

Create a Scene by instantiating the class named **Scene** which belongs to the package **javafx.scene.** To this class pass the Group object (**root**) created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 6: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object which is passed to the start method of the scene class, as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 7: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using this method as follows.

```
primaryStage.setScene(scene);
```

Step 8: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 9: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows.

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

Example

Following is a program which generates a quadrilateral curve using JavaFX. Save this code in a file with the name **QuadCurveExample.java.**

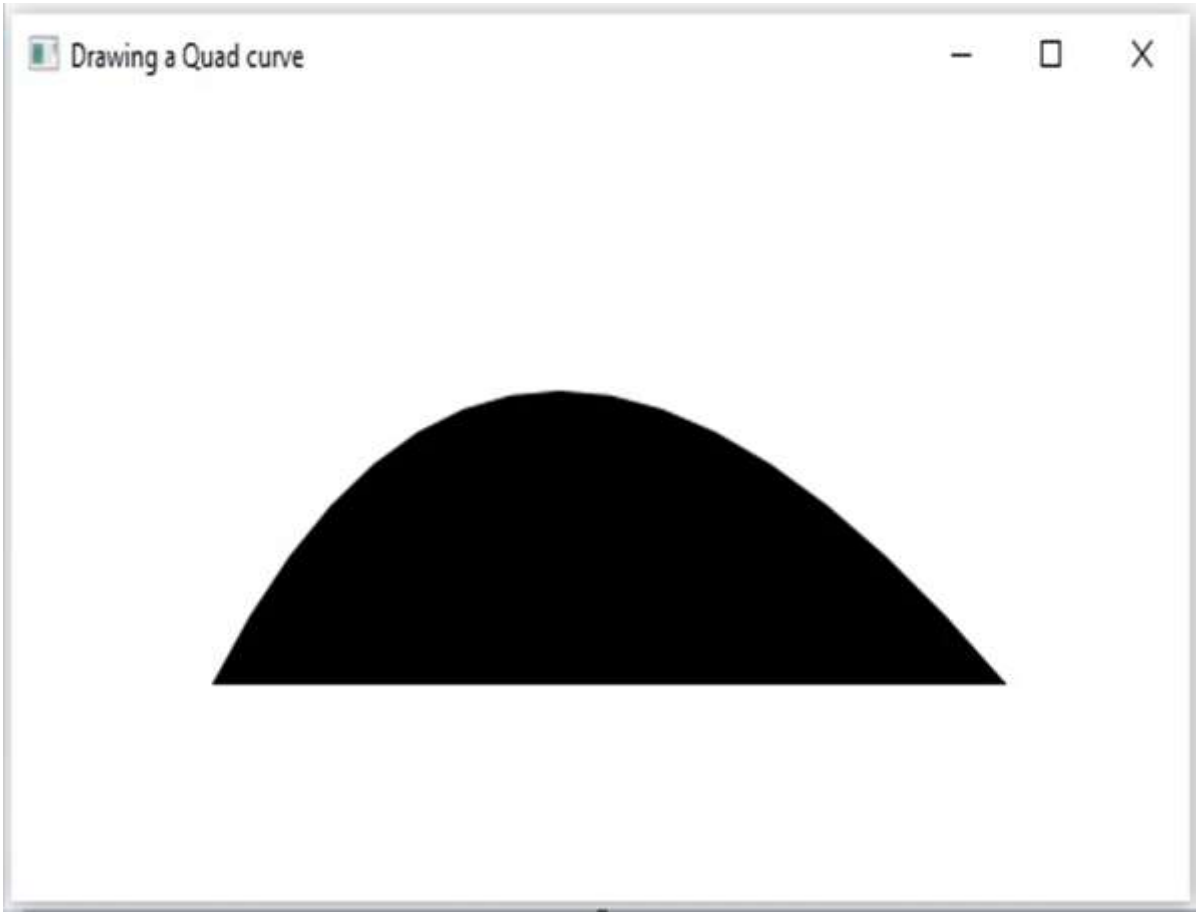
```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.shape.QuadCurve;

public class QuadCurveExample extends Application {

   @Override
   public void start(Stage stage) {

      //Creating a QuadCurve
      QuadCurve quadCurve = new QuadCurve();

      //Adding properties to the Quad Curve
      quadCurve.setStartX(100.0);
      quadCurve.setStartY(220.0f);
      quadCurve.setEndX(500.0f);
      quadCurve.setEndY(220.0f);
      quadCurve.setControlX(250.0f);
      quadCurve.setControlY(0.0f);

      //Creating a Group object
      Group root = new Group(quadCurve);
```

```
      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Drawing a Quad curve");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac QuadCurveExample.java
java QuadCurveExample
```

On executing, the above program generates a JavaFX window displaying a Bezier quadrilateral curve as shown in the following screenshot.



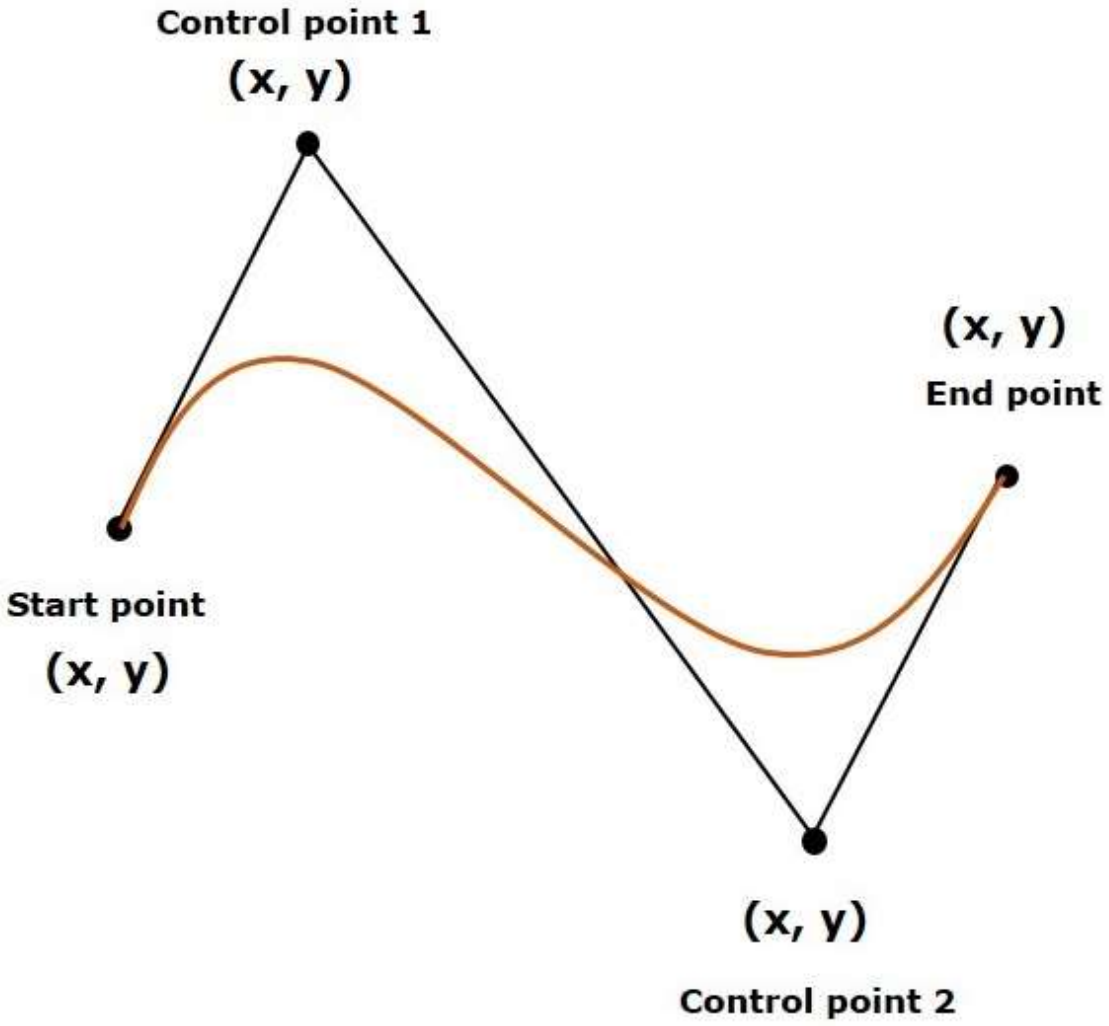
2D Shapes – CubicCurve

A CubicCurve is described by a third-degree polynomial function of two variables, and can be written in the following form –

$$a_1x^3 + a_2x^2y + a_3xy^2 + a_4y^3 + a_5x^2 + a_6xy + a_7y^2 + a_8x + a_9y + a_{10} = 0.$$

These Bezier curves are generally used in computer graphics. They are parametric curves which appear reasonably smooth at all scales. These curves are drawn based on points on the XY plane.

A cubic curve is a Bezier parametric curve in the XY plane is a curve of degree 3. It is drawn using four points: **Start Point**, **End Point**, **Control Point** and **Control Point2** as shown in the following diagram.



In JavaFX, a CubicCurve is represented by a class named **CubicCurve.** This class belongs to the package **javafx.scene.shape**.

By instantiating this class, you can create a CubicCurve node in JavaFX.

This class has 8 properties of the double datatype namely –

- **startX:** The x coordinate of the starting point of the curve.
- **startY:** The y coordinate of the starting point of the curve.
- **controlX1:** The x coordinate of the first control point of the curve.
- **controlY1:** The y coordinate of the first control point of the curve.
- **controlX2:** The x coordinate of the second control point of the curve.
- **controlY2:** The y coordinate of the second control point of the curve.
- **endX:** The x coordinate of the end point of the curve.
- **endY:** The y coordinate of the end point of the curve.

To draw a cubic curve, you need to pass values to these properties, either by passing them to the constructor of this class, in the same order, at the time of instantiation, as shown below –

```
CubicCurve cubiccurve = new CubicCurve(startX, startY, controlX1, controlY1,
controlX2, controlY2, endX, endY);
```

Or, by using their respective setter methods as follows –

```
setStartX(value);
setStartY(value);
setControlX1(value);
setControlY1(value);
setControlX2(value);
setControlY2(value);
setEndX(value);
setEndY(value);
```

To draw a Bezier cubic curve in JavaFX, follow the steps given below.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application** and implement the **start()** method of this class as follows.

```
public class ClassName extends Application {


    @Override
    public void start(Stage primaryStage) throws Exception {


    }


}
```

Step 2: Creating a CubicCurve

You can create a CubicCurve in JavaFX by instantiating the class named **CubicCurve** which belongs to a package **javafx.scene.shape.** You can instantiate this class as follows.

```
//Creating an object of the class CubicCurve
CubicCurve cubiccurve = new CubicCurve();
```

Step 3: Setting Properties to the CubicCurve

Specify the x, y coordinates of the four points: **start point**, **end point**, **control point1** and **control point2** of the required curve, using their respective setter methods as shown in the following code block.

```
//Setting properties to cubic curve
cubicCurve.setStartX(100.0f);
cubicCurve.setStartY(150.0f);
cubicCurve.setControlX1(400.0f);
cubicCurve.setControlY1(40.0f);
cubicCurve.setControlX2(175.0f);
cubicCurve.setControlY2(250.0f);
cubicCurve.setEndX(500.0f);
cubicCurve.setEndY(150.0f);
```

Step 4: Creating a Group Object

In the **start()** method, create a group object by instantiating the class named **Group**, which belongs to the package **javafx.scene**.

Pass the CubicCurve (node) object created in the previous step as a parameter to the constructor of the Group class in order to add it to the group as follows –

```
Group root = new Group(cubiccurve);
```

Step 5: Creating a Scene Object

Create a Scene by instantiating the class named **Scene** which belongs to the package **javafx.scene.** To this class pass the Group object (**root**) created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 6: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object which is passed to the start method of the scene class as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 7: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using this method as follows.

```
primaryStage.setScene(scene);
```

Step 8: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 9: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows.

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

Example

Following is a program which generates a Bezier CubicCurve using JavaFX. Save this code in a file with the name **CubicCurveExample.java.**

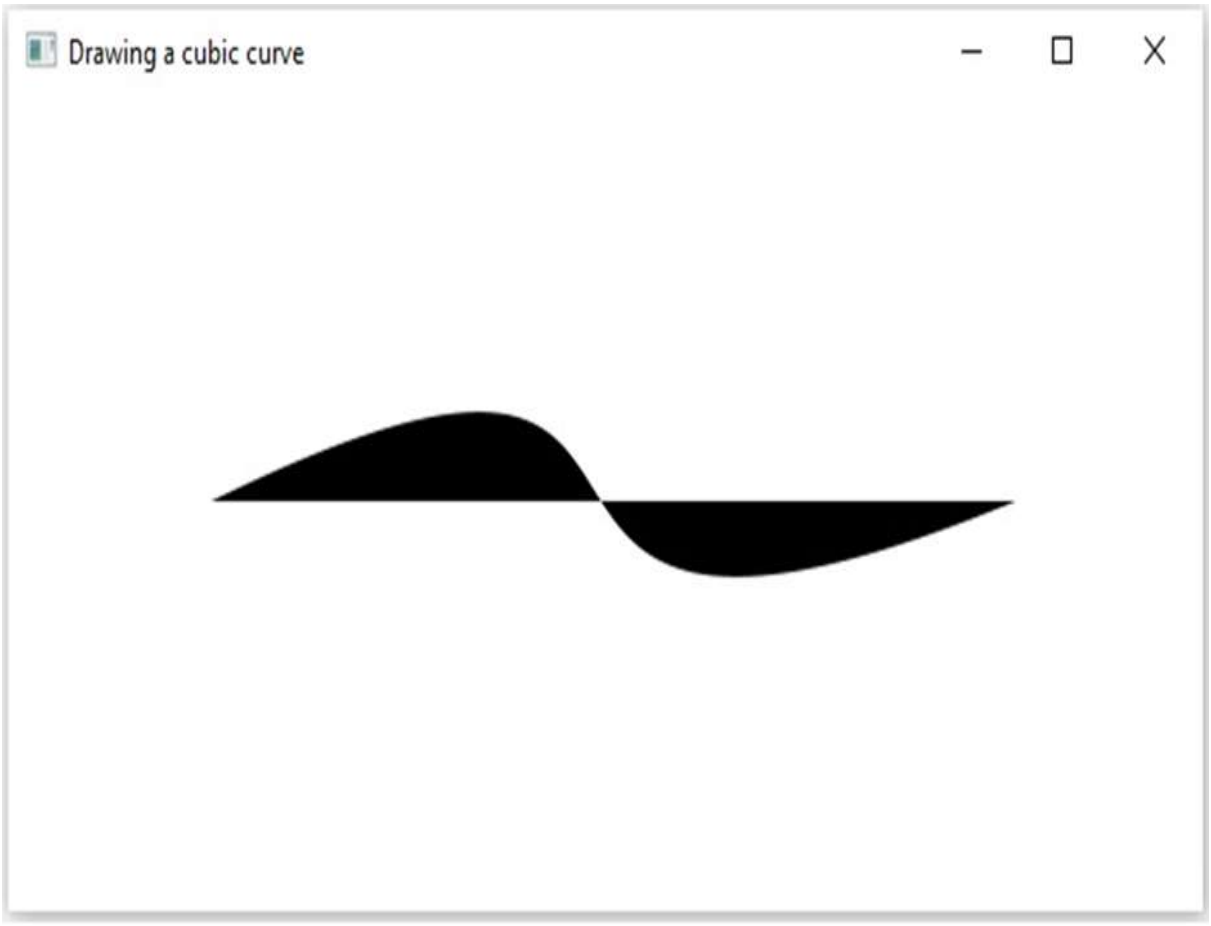
```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.shape.CubicCurve;

public class CubicCurveExample extends Application {

   @Override
   public void start(Stage stage) {

      //Drawing a cubic curve
      CubicCurve cubicCurve = new CubicCurve();

      //Setting properties to cubic curve
      cubicCurve.setStartX(100.0f);
```

```
      cubicCurve.setStartY(150.0f);
      cubicCurve.setControlX1(400.0f);
      cubicCurve.setControlY1(40.0f);
      cubicCurve.setControlX2(175.0f);
      cubicCurve.setControlY2(250.0f);
      cubicCurve.setEndX(500.0f);
      cubicCurve.setEndY(150.0f);

      //Creating a Group object
      Group root = new Group(cubicCurve);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Drawing a cubic curve");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

      }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac CubicCurveExample.java
java CubicCurveExample
```

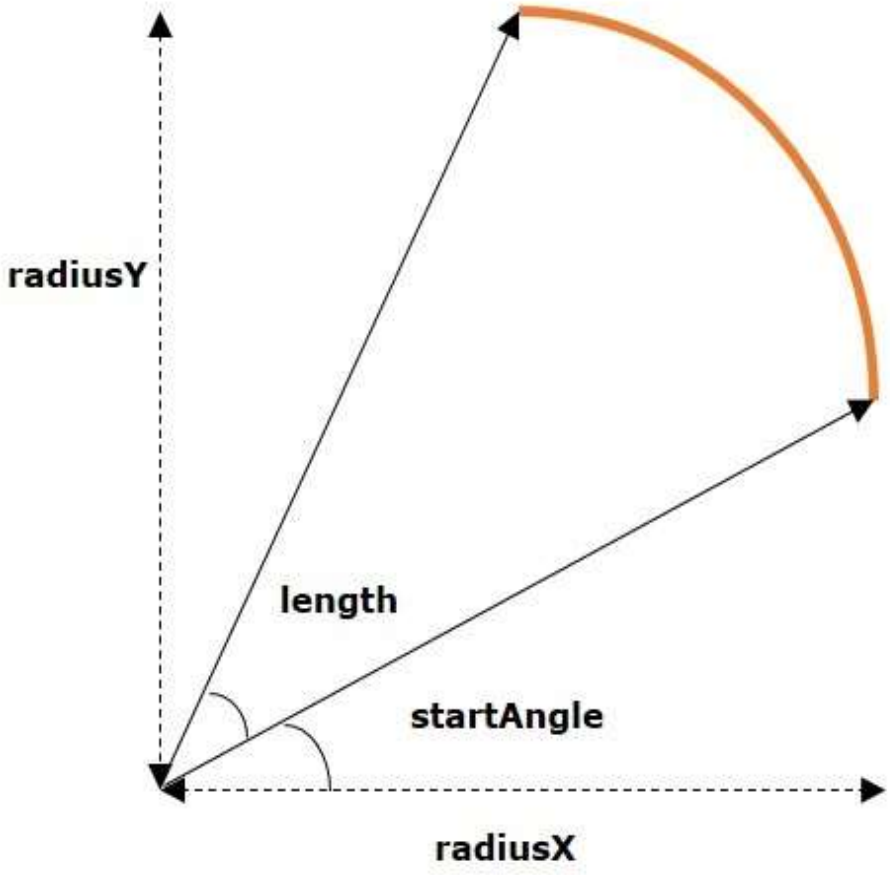
On executing, the above program generates a JavaFX window displaying a Bezier cubic curve as shown below.



2D Shapes – Arc

An arc is part of a curve. It is described by the following properties —

- **length:** The distance along the arc.
- **angle**: The angle the curve makes at the centre of the circle.
- **radiusX:** The width of the full Ellipse of which the current arc is a part of.
- **radiusY:** The height of the full Ellipse of which the current arc is a part of.



In JavaFX, an arc is represented by a class named **Arc.** This class belongs to the package **javafx.scene.shape**.

By instantiating this class, you can create an arc node in JavaFX.

This class has a few properties of the double datatype namely –

- **centerX:** The x coordinate of the center of the arc.
- **centerY:** The y coordinate of the center of the arc.
- **radiusX:** The width of the full ellipse of which the current arc is a part of.
- **radiusY:** The height of the full ellipse of which the current arc is a part of.
- **startAngle:** The starting angle of the arc in degrees.
- **length:** The angular extent of the arc in degrees.

To draw an arc, you need to pass values to these properties, either by passing them to the constructor of this class, in the same order, at the time of instantiation, as shown below –

```
Circle circle = new Circle(centerX, centerY, radiusX, radiusY);
```

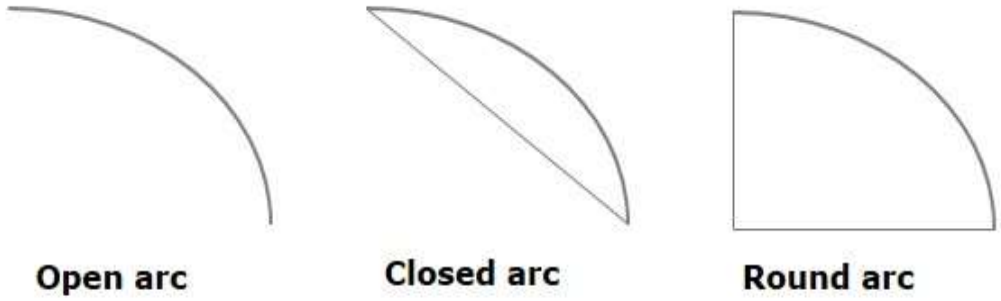
Or, by using their respective setter methods as follows –

```
setCenterX(value);
setCenterY(value);
setRadiusX(value);
setRadiusY(value);
```

Types of Arc:

In JavaFX, you can draw three kinds of arc's namely –

- **Open:** An arc which is not closed at all is known as an open arc.
- **Chord:** A chord is a type of an arc which is closed by straight line.
- **Round:** The Round arc is an arc which is closed by joining the starting and ending point to the center of the ellipse.



You can set the type of the arc using the method **setType()** by passing any of the following properties: **ArcType.OPEN**, **ArcType.CHORD**, **ArcType.ROUND**.

To Draw an arc in JavaFX, follow the steps given below.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application** and implement the **start()** method of this class as follows.

```
public class ClassName extends Application {


    @Override
    public void start(Stage primaryStage) throws Exception {


    }


}
```

Step 2: Creating an Arc

You can create an arc in JavaFX by instantiating the class named **Arc** which belongs to a package **javafx.scene.shape.** You can instantiate this class as shown below.

```
//Creating an object of the class Arc
Arc arc = new Arc();
```

Step 3: Setting Properties to the Arc

Specify the x, y coordinates of the center of the Ellipse (of which this arc is a part of). These coordinates include – radiusX, radiusY, start angle and length of the arc using their respective setter methods as shown in the following code block.

You can also set the type of the arc (round, chord open) by using the **setType()** method.

```
//Setting the properties of the arc
arc.setCenterX(300.0f);
arc.setCenterY(150.0f);
arc.setRadiusX(90.0f);
arc.setRadiusY(90.0f);
arc.setStartAngle(40.0f);
arc.setLength(239.0f);
arc.setType(ArcType.ROUND);
```

Step 4: Setting the Type of the Arc

You can set the type of the arc using the method **setType()** as shown in the following code block.

```
//Setting the type of the arc
arc.setType(ArcType.ROUND);
```

Step 5: Creating a Group Object

In the **start()** method create a group object by instantiating the class named **Group**, which belongs to the package **javafx.scene**.

Pass the Arc (node) object created in the previous step as a parameter to the constructor of the Group class. This should be done in order to add it to the group as shown below –

```
Group root = new Group(arc);
```

Step 6: Creating a Scene Object

Create a Scene by instantiating the class named **Scene** which belongs to the package **javafx.scene.** To this class pass the Group object (**root**) created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 7: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object which is passed to the start method of the scene class as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 8: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using this method as follows.

```
primaryStage.setScene(scene);
```

Step 9: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 10: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows.

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

Example

Following is a program which generates an arc. Save this code in a file with the name **ArcExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
```

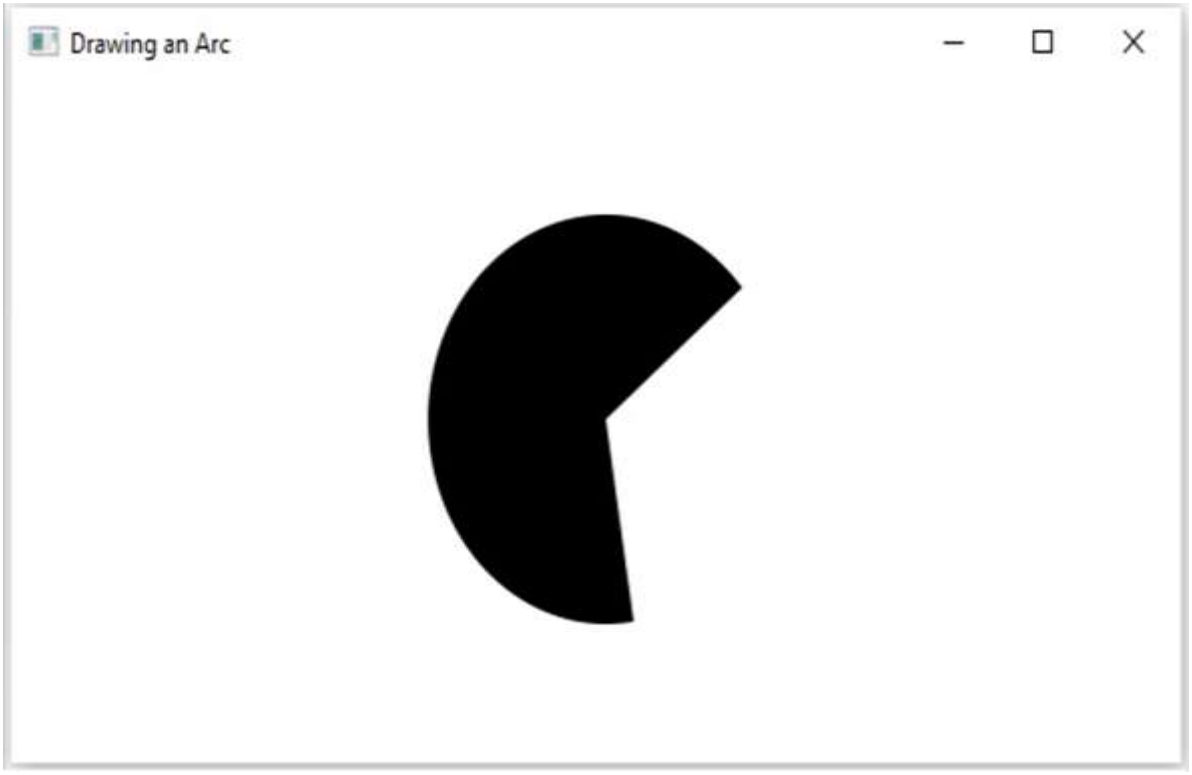
```
import javafx.stage.Stage;
import javafx.scene.shape.Arc;

import javafx.scene.shape.ArcType;

public class ArcExample extends Application {
   @Override
   public void start(Stage stage) {

      //Drawing an arc
      Arc arc = new Arc();

      //Setting the properties of the arc
      arc.setCenterX(300.0f);
      arc.setCenterY(150.0f);
      arc.setRadiusX(90.0f);
      arc.setRadiusY(90.0f);
      arc.setStartAngle(40.0f);
      arc.setLength(239.0f);

      //Setting the type of the arc
      arc.setType(ArcType.ROUND);

      //Creating a Group object
      Group root = new Group(arc);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Drawing an Arc");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
```

```
      stage.show();

   }

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

Compile and execute the saved Java file from the command prompt using the following commands.

```
javac ArcExample.java
java ArcExample
```

On executing, the above program generates a JavaFX window displaying an arc as shown in the following screenshot.



2D Shapes – SVGPath

SVG **(Scalable Vector Graphics)** is an XML based language to define vector based graphics.

In JavaFX we can construct images by parsing SVG paths. Such shapes are represented by the class named **SVGPath.** This class belongs to the package **javafx.scene.shape**.

By instantiating this class, you can create a node which is created by parsing an SVG path in JavaFX.

This class has a property named **content** of String datatype. This represents the SVG Path encoded string, from which the image should be drawn.

To draw a shape by parsing an SVG path, you need to pass values to this property, using the method named **setContent()** of this class as follows –

```
setContent(value);
```

To Draw a shape by parsing an SVGPath in JavaFX, follow the steps given below.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application** and implement the **start()** method of this class as follows.

```
public class ClassName extends Application {


    @Override
    public void start(Stage primaryStage) throws Exception {


    }


}
```

Step 2: Creating an Object of the SVGPath Class

You can create a required shape in JavaFX by parsing an SVGPath. To do so, instantiate the class named **SVGPath** which belongs to a package **javafx.scene.shape.** You can instantiate this class as follows.

```
//Creating an object of the class SVGPath
SVGPath svgpath = new SVGPath();
```

Step 3: Setting the SVGPath

Set the content for the SVG object using the method **setContent()**. To this method, you need to pass the SVGPath. Using which, a shape should be drawn in the form of a string as shown in the following code block.

```
String path = "M 100 100 L 300 100 L 200 300 z";
```

```
//Setting the SVGPath in the form of string
svgPath.setContent(path);
```

Step 4: Creating a Group Object

In the **start()** method, create a group object by instantiating the class named **Group**, which belongs to the package **javafx.scene**.

Pass the SVGPath (node) object created in the previous step as a parameter to the constructor of the Group class. This should be done in order to add it to the group as follows -

```
Group root = new Group(svgpath);
```

Step 5: Creating a Scene Object

Create a Scene by instantiating the class named **Scene** which belongs to the package **javafx.scene.** To this class pass the Group object (**root**) created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 6: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object which is passed to the start method of the scene class as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 7: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using this method as follows.

```
primaryStage.setScene(scene);
```

Step 8: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 9: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main x method as follows.

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

Example

Following is a program which generates a shape by parsing SVG path using JavaFX. Save this code in a file with the name **SVGExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.shape.SVGPath;
import javafx.stage.Stage;

public class SVGExample extends Application {

   @Override
   public void start(Stage stage) {

      //Creating a SVGPath object
      SVGPath svgPath = new SVGPath();

      String path = "M 100 100 L 300 100 L 200 300 z";

      //Setting the SVGPath in the form of string
      svgPath.setContent(path);

      //Creating a Group object
      Group root = new Group(svgPath);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);
```

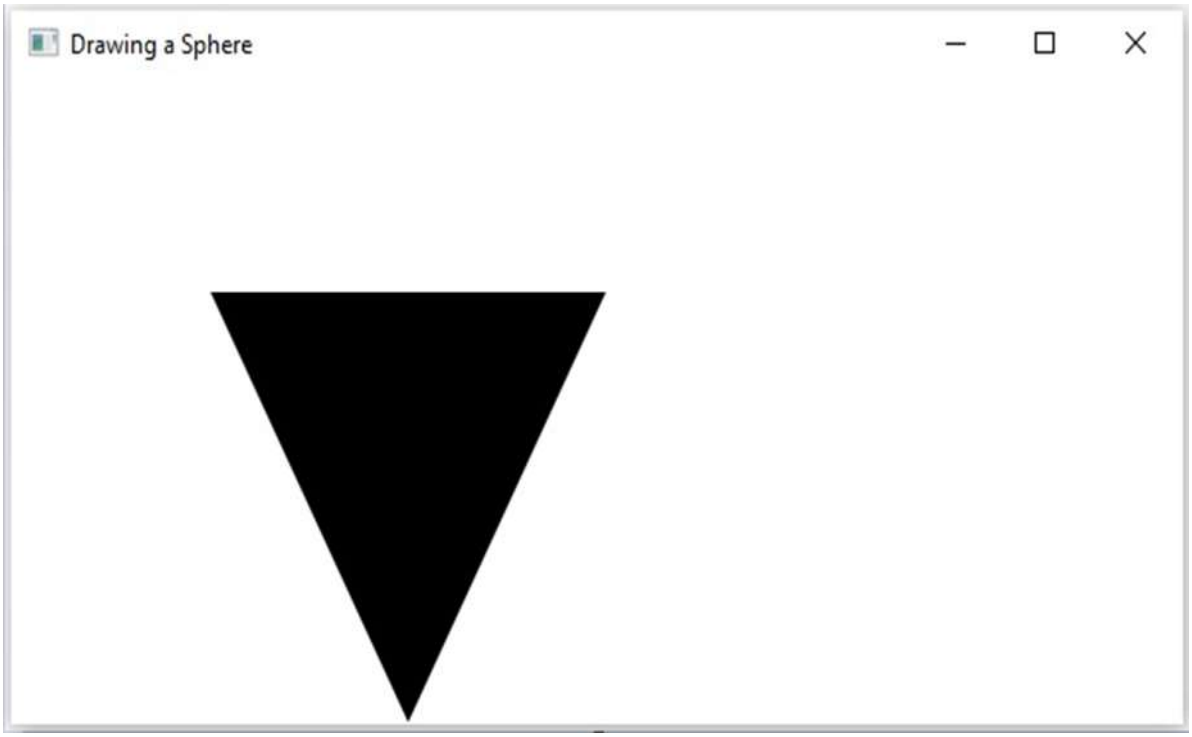
```
      //Setting title to the Stage
```

```
      stage.setTitle("Drawing a Sphere");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }

   public static void main(String args[]){

      launch(args);

   }

}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac SVGExample.java
java SVGExample
```

On executing, the above program generates a JavaFX window displaying a triangle, which is drawn by parsing the **SVG path** as shown below.

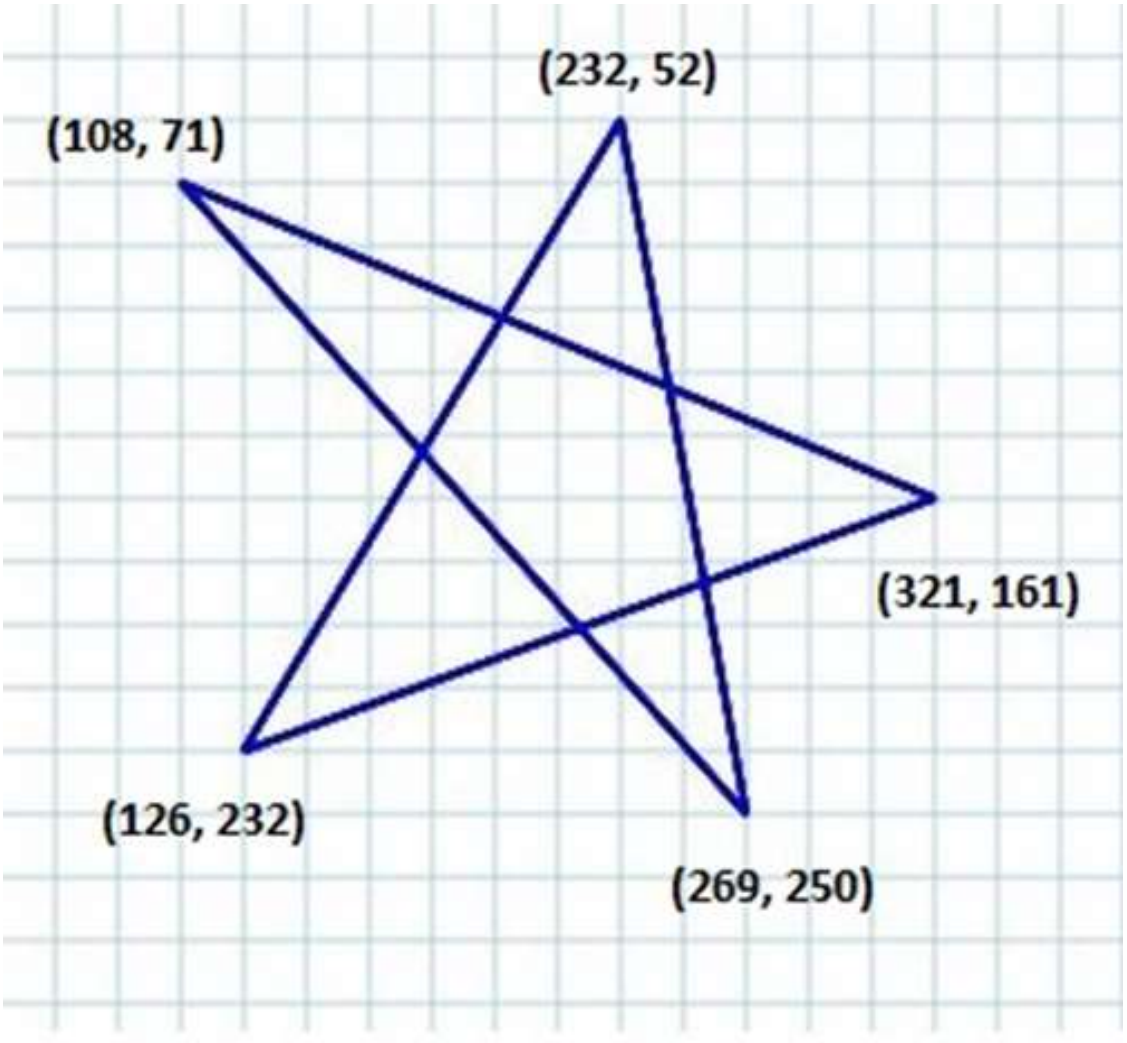


Drawing More Shapes Through Path Class

In the previous section, we have seen how to draw some simple predefined shapes by instantiating classes and setting respective parameters.

But, just these predefined shapes are not sufficient to build complexer shapes other than the primitives provided by the **javafx.shape** package.

For example, if you want to draw a graphical element as shown in the following diagram, you cannot rely on those simple shapes.



The Path Class

To draw such complex structures JavaFX provides a class named **Path.** This class represents the geometrical outline of a shape.

It is attached to an observable list which holds various **Path Elements** such as moveTo, LineTo, HlineTo, VlineTo, ArcTo, QuadCurveTo, CubicCurveTo.

On instantiating, this class constructs a path based on the given path elements.

You can pass the path elements to this class while instantiating it as follows —

```
Path myshape = new Path(pathElement1, pathElement2, pathElement3);
```

Or, you can get the observable list and add all the path elements using **addAll()** method as follows —

```
Path myshape = new Path();
myshape.getElements().addAll(pathElement1, pathElement2, pathElement3);
```

You can also add elements individually using the add() method as –

```
Path myshape = new Path();
myshape.getElements().add(pathElement1);
```

The Move to Path Element

The Path Element **MoveTo** is used to move the current position of the path to a specified point. It is generally used to set the starting point of a shape drawn using the path elements.

It is represented by a class named **LineTo** of the package **javafx.scene.shape**. It has 2 properties of the double datatype namely –

- **X:** The x coordinate of the point to which a line is to be drawn from the current position.
- **Y:** The y coordinate of the point to which a line is to be drawn from the current position.

You can create a move to path element by instantiating the MoveTo class and passing the x, y coordinates of the new point as follows –

```
MoveTo moveTo = new MoveTo(x, y);
```

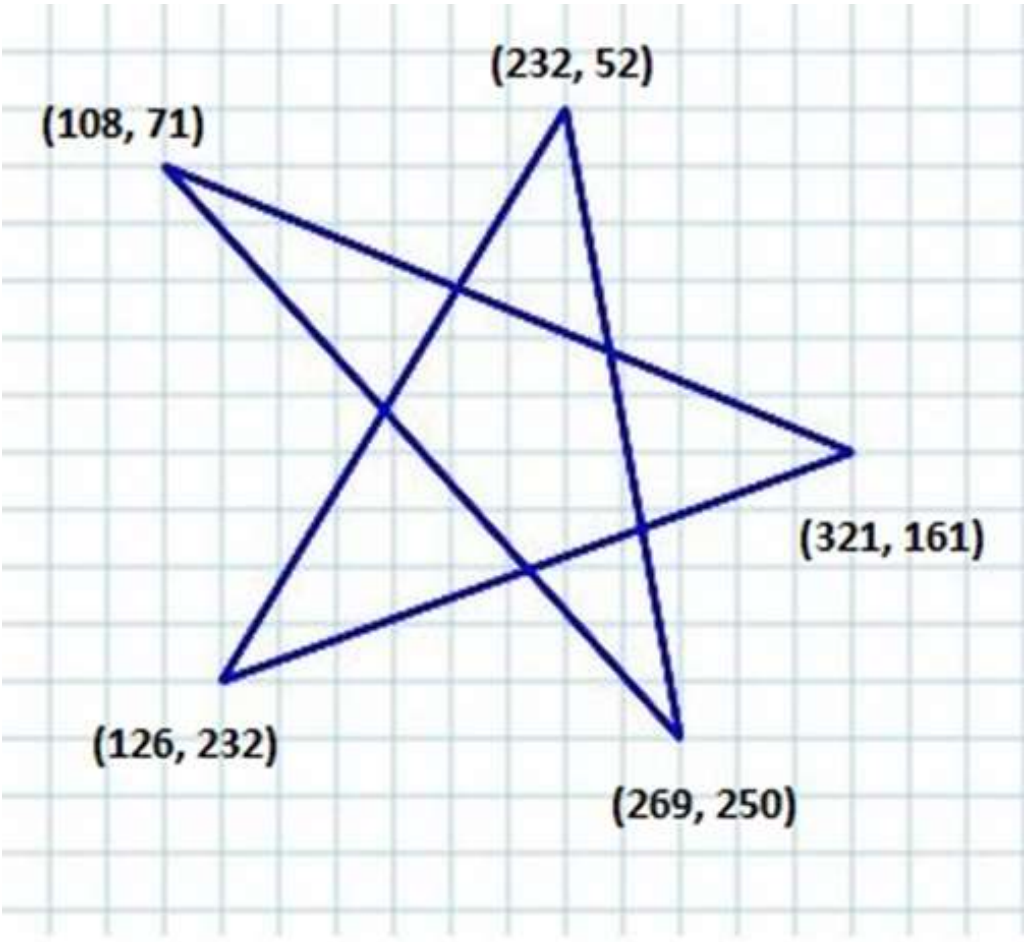
If you don't pass any values to the constructor, then the new point will be set to (0,0).

You can also set values to the x, y coordinate, using their respective setter methods as follows –

```
setX(value);
setY(value);
```

Example – Drawing a Complex Path

In this example, we will show how to draw the following shape using the **Path, MoveTo** and **Line** classes.



Save this code in a file with the name **ComplexShape.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.shape.LineTo;
import javafx.scene.shape.MoveTo;
import javafx.scene.shape.Path;

public class ComplexChape extends Application {
   @Override
   public void start(Stage stage) {
```

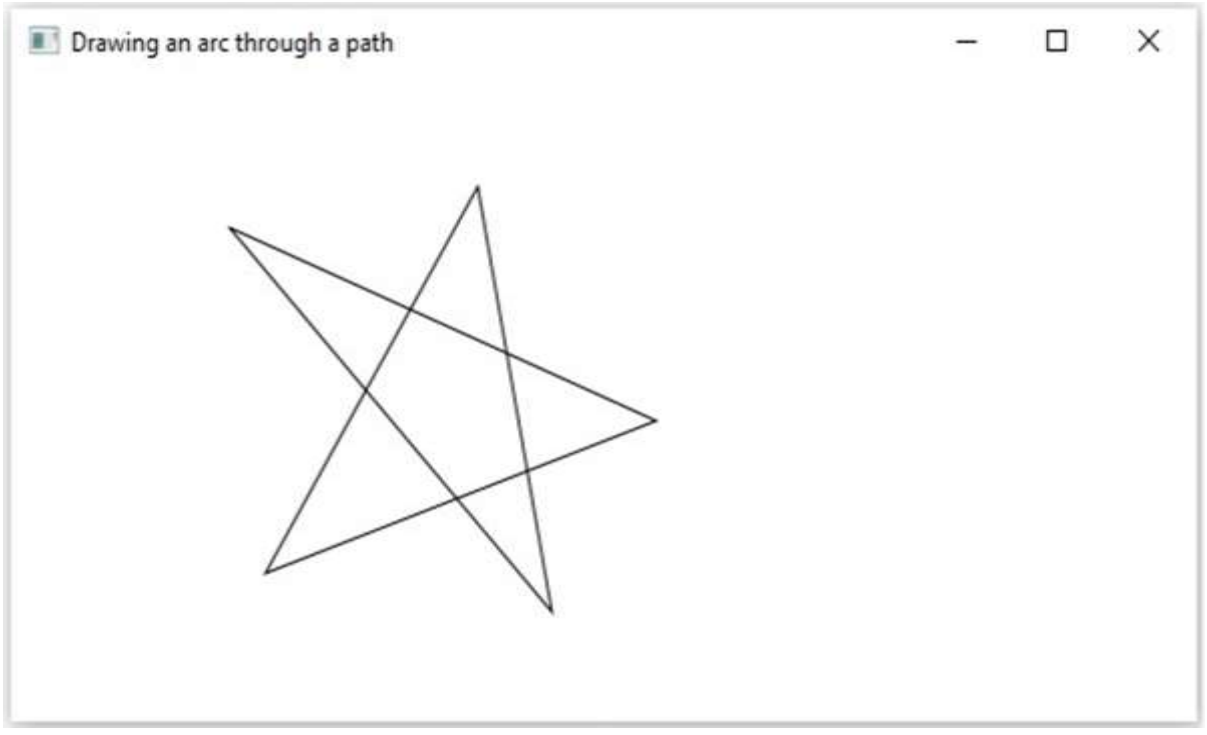
```
//Creating a Path
Path path = new Path();

//Moving to the starting point
MoveTo moveTo = new MoveTo(108, 71);

//Creating 1st line
LineTo line1 = new LineTo(321, 161);

//Creating 2nd line
LineTo line2 = new LineTo(126,232);

//Creating 3rd line
LineTo line3 = new LineTo(232,52);

//Creating 4th line
LineTo line4 = new LineTo(269, 250);

//Creating 5th line
LineTo line5 = new LineTo(108, 71);

//Adding all the elements to the path
path.getElements().add(moveTo);
path.getElements().addAll(line1, line2, line3, line4, line5);

//Creating a Group object
Group root = new Group(path);

//Creating a scene object
Scene scene = new Scene(root, 600, 300);

//Setting title to the Stage
stage.setTitle("Drawing an arc through a path");

//Adding scene to the stage
stage.setScene(scene);
```

```
      //Displaying the contents of the stage
      stage.show();
   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac ComplexShape.java
java ComplexShape
```

On executing, the above program generates a JavaFX window displaying an arc, which is drawn from the current position to the specified point as shown below.



Following are the various path elements (classes) provided by JavaFX. These classes exist in the package **javafx.shape**. All these classes inherit the class **PathElement**.

S. No.	Shape and Description
1	**LineTo:** The path element **line** is used to draw a straight line to a point in the specified coordinates from the current position. It is represented by a class named **LineTo.** This class belongs to the package **javafx.scene.shape**. This class has 2 properties of the double datatype namely – • **X:** The x coordinate of the point to which a line is to be drawn from the current position. • **Y:** The y coordinate of the point to which a line is to be drawn from the current position.
2	**HLineTo:** The path element **HLineTo** is used to draw a horizontal line to a point in the specified coordinates from the current position. It is represented by a class named **HLineTo**. This class belongs to the package **javafx.scene.shape**. This class has a properties of the double datatype namely – • **X:** The x coordinate of the point to which a horizontal line is to be drawn from the current position.
3	**VLineTo:** The path element **vertical line** is used to draw a vertical line to a point in the specified coordinates from the current position. It is represented by a class named **VLineTo.** This class belongs to the package **javafx.scene.shape**. This class has a property of the double datatype namely – • **Y:** The y coordinate of the point to which a vertical is to be drawn from the current position.

4	**QuadCurveTo:** The path element quadratic curve is used to draw a **quadratic curve** to a point in the specified coordinates from the current position. It is represented by a class named **QuadraticCurveTo.** This class belongs to the package **javafx.scene.shape**. This class has 4 properties of the double datatype namely – • **setX:** The x coordinate of the point to which a curve is to be drawn from the current position. • **setX:** The y coordinate of the point to which a curve is to be drawn from the current position. • **controlX:** The x coordinate of the control point of the curve. • **controlY:** The y coordinate of the control point of the curve.
5	**CubicCurveTo:** The path element cubic curve is used to draw a **cubic curve** to a point in the specified coordinates from the current position. It is represented by a class named **CubicCurveTo.** This class belongs to the package **javafx.scene.shape**. This class has 6 properties of the double datatype namely – • **setX:** The x coordinate of the point to which a curve is to be drawn from the current position. • **setX:** The y coordinate of the point to which a curve is to be drawn from the current position. • **controlX1:** The x coordinate of the 1st control point of the curve. • **controlY1:** The y coordinate of the 1st control point of the curve. • **controlX2:** The x coordinate of the 2nd control point of the curve. • **controlY2:** The y coordinate of the 2nd control point of the curve.

6	**ArcTo:** The path element **Arc** is used to draw an arc to a point in the specified coordinates from the current position. It is represented by a class named **ArcTo.** This class belongs to the package **javafx.scene.shape**. This class has 4 properties of the double datatype namely – • **X:** The x coordinate of the center of the arc. • **Y:** The y coordinate of the center of the arc. • **radiusX:** The width of the full ellipse of which the current arc is a part of. • **radiusY:** The height of the full ellipse of which the current arc is a part of.

PathElement – Line

The path element **line** is used to draw a straight line to a point in the specified coordinates from the current position.

It is represented by a class named **LineTo.** This class belongs to the package **javafx.scene.shape**.

This class has 2 properties of the double datatype namely –

- **X:** The x coordinate of the point to which a line is to be drawn from the current position.
- **Y:** The y coordinate of the point to which a line is to be drawn from the current position.

To draw a line, you need to pass values to these properties. This can be either done by passing them to the constructor of this class, in the same order, at the time of instantiation, as shown below –

```
LineTO line = new LineTo(x, y);
```

Or, by using their respective setter methods as follows –

```
setX(value);
setY(value);
```

To draw a line to a specified point from the current position in JavaFX, follow the steps given below.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application** and implement the **start()** method of this class as follows.

```
public class ClassName extends Application {


    @Override
    public void start(Stage primaryStage) throws Exception {
    }
}
```

Step 2: Create the Path Class Object

You can create the path class object as shown below.

```
//Creating a Path object
Path path = new Path();
```

Step 3: Setting the Path

Create the **MoveTo** path element and set the XY coordinates to starting point of the line to the coordinates (100, 150). This can be done using the methods **setX()** and **setY()** of the class **MoveTo** as shown below.

```
//Moving to the starting point
MoveTo moveTo = new MoveTo();
moveTo.setX(100.0f);
moveTo.setY(150.0f);
```

Step 4: Creating an Object of the Class LineTo

Create the path element line by instantiating the class named **LineTo** which belongs to the package **javafx.scene.shape** as follows.

```
//Creating an object of the class LineTo
LineTo lineTo = new LineTo();
```

Step 5: Setting Properties to the Line Element

Specify the coordinates of the point to which a line is to be drawn from the current position. This can be done by setting the properties x and y using their respective setter methods as shown in the following code block.

```
//Setting the Properties of the line element
lineTo.setX(500.0f);
lineTo.setY(150.0f);
```

Step 6: Adding Elements to the Observable List of the Path Class

Add the path elements **MoveTo** and **LineTo** created in the previous steps to the observable list of the **Path** class as shown below —

```
//Adding the path elements to Observable list of the Path class
path.getElements().add(moveTo);
path.getElements().add(lineTo);
```

Step 7: Creating a Group Object

Create a group object by instantiating the class named **Group**, which belongs to the package **javafx.scene**.

Pass the Line (node) object created in the previous step as a parameter to the constructor of the Group class. This should be done in order to add it to the group as shown below –

```
Group root = new Group(line);
```

Step 8: Creating a Scene Object

Create a Scene by instantiating the class named **Scene** which belongs to the package **javafx.scene.** To this class, pass the Group object (**root**) created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 9: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object which is passed to the start method of the scene class, as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 10: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using this method as shown below –

```
primaryStage.setScene(scene);
```

Step 11: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 12: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows –

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

Example

The following program shows how to draw a straight line from the current point to a specified position using the class Path of JavaFX. Save this code in a file with the name **LineToExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.shape.LineTo;
import javafx.scene.shape.MoveTo;
import javafx.scene.shape.Path;
import javafx.stage.Stage;

public class LineToExample extends Application {

   @Override
   public void start(Stage stage) {
```

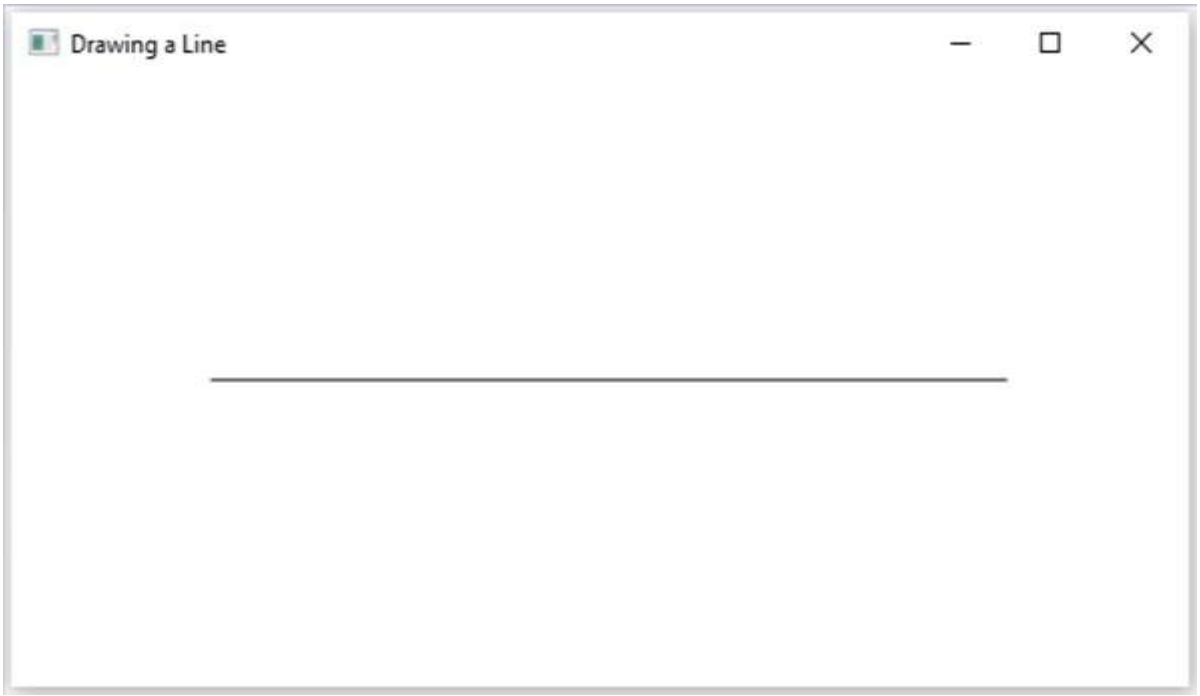
```
        //Creating a Path object
        Path path = new Path();

        //Moving to the starting point
        MoveTo moveTo = new MoveTo();
        moveTo.setX(100.0f);
        moveTo.setY(150.0f);

        //Instantiating the LineTo class

        LineTo lineTo = new LineTo();

        //Setting the Properties of the line element
        lineTo.setX(500.0f);
        lineTo.setY(150.0f);

        //Adding the path elements to Observable list of the Path class
        path.getElements().add(moveTo);
        path.getElements().add(lineTo);

        //Creating a Group object
        Group root = new Group(path);

        //Creating a scene object
        Scene scene = new Scene(root, 600, 300);

        //Setting title to the Stage
        stage.setTitle("Drawing a Line");

        //Adding scene to the satge
        stage.setScene(scene);

        //Displaying the contents of the stage
        stage.show();

    }
```

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

      launch(args);

   }
}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac LineToExample.java
java LineToExample
```

On executing, the above program generates a JavaFX window displaying a straight line, which is drawn from the current position to the specified point, as shown below.



PathElement – Horizontal Line

The path element **HLineTo** is used to draw a horizontal line to a point in the specified coordinates from the current position.

It is represented by a class named **HLineTo**. This class belongs to the package **javafx.scene.shape**.

This class has a property of the double datatype namely –

- **X:** The x coordinate of the point to which a horizontal line is to be drawn from the current position.

To draw a path element horizontal line, you need to pass a value to this property. This can be either done by passing it to the constructor of this class, at the time of instantiation, as follows –

```
HLineTO hline = new HLineTo(x);
```

Or, by using its respective setter methods as shown below –

```
setX(value);
```

Follow the steps given below to draw a horizontal line to a specified point from the current position in JavaFX.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application** and implement the **start()** method of this class as follows.

```
public class ClassName extends Application {


    @Override
    public void start(Stage primaryStage) throws Exception {


    }


}
```

Step 2: Instantiating the Path Class

Create the path class object as follows –

```
//Creating a Path object
Path path = new Path();
```

Step 3: Setting the Initial Point

Create the **MoveTo** path element and set XY coordinates to starting point of the line to the coordinates (100, 150). This can be done by using the methods **setX()** and **setY()** of the class **MoveTo** as shown below.

```
//Moving to the starting point
MoveTo moveTo = new MoveTo();
```

```
moveTo.setX(100.0f);
moveTo.setY(150.0f);
```

Step 4: Creating an Object of the Class HLineTo

Create the path element horizontal line by instantiating the class named **HLineTo** which belongs to the package **javafx.scene.shape** as shown below.

```
//Creating an object of the class HLineTo
HLineTo hLineTo = new HLineTo();
```

Step 5: Setting Properties to the Horizontal Line Element

Specify the x coordinate of the point to which a horizontal line is to be drawn from the current position. This can be done by setting the property x, using the method **setX()** of the **HLineTo** class as shown below.

```
//Setting the Properties of the horizontal line element
hlineTo.setX(400);
```

Step 6: Adding Elements to the Observable List of Path Class

Add the path elements **MoveTo** and **HlineTo** created in the previous steps to the observable list of the **Path** class as shown below —

```
//Adding the path elements to Observable list of the Path class
path.getElements().add(moveTo);
path.getElements().add(hlineTo);
```

Step 7: Creating a Group Object

Create a group object by instantiating the class named **Group**, which belongs to the package **javafx.scene**.

Pass the Line (node) object created in the previous step as a parameter to the constructor of the Group class. This should be done in order to add it to the group as shown below —

```
Group root = new Group(line);
```

Step 8: Creating a Scene Object

Create a Scene by instantiating the class named **Scene** which belongs to the package **javafx.scene.** To this class, pass the Group object (**root**) created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen along with the object of the Group class as follows —

```
Scene scene = new Scene(group ,600, 300);
```

Step 9: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object which is passed to the start method of the scene class as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 10: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using this method as shown below —

```
primaryStage.setScene(scene);
```

Step 11: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 12: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows.

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

Example

Following is a program which draws a horizontal line from the current point to a specified position using the class **Path** of JavaFX. Save this code in a file with the name – **HLineToExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.shape.HLineTo;
import javafx.scene.shape.MoveTo;
```

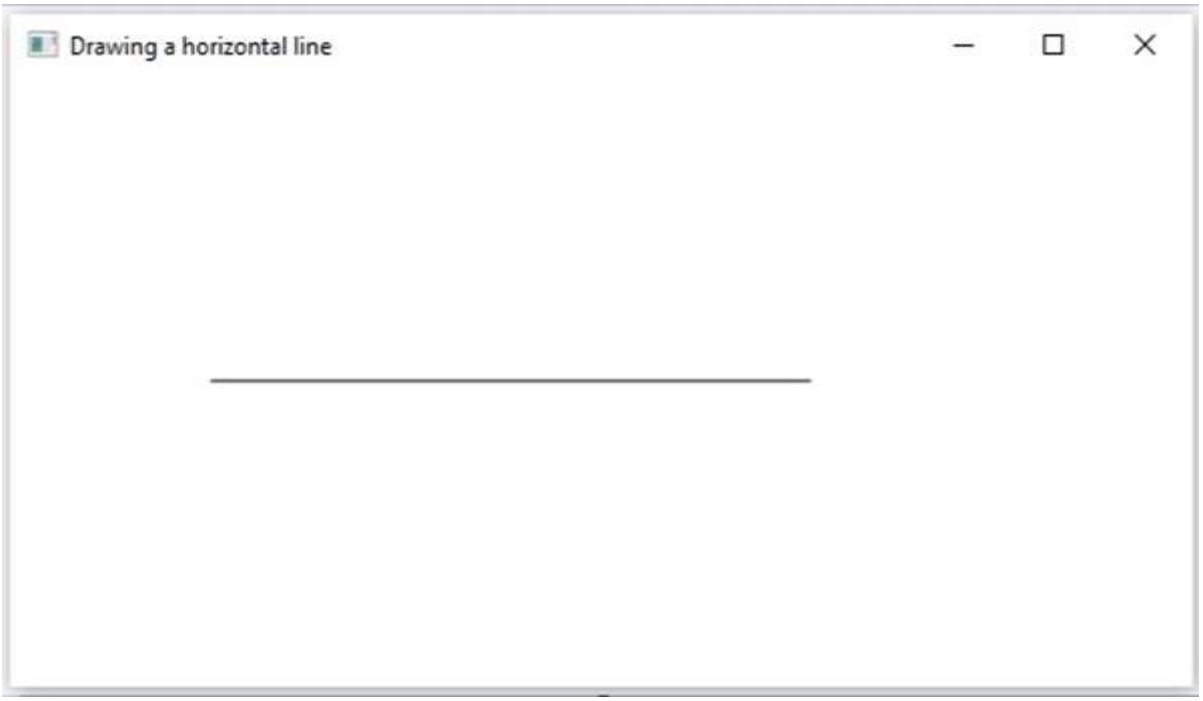
```
import javafx.scene.shape.Path;
public class HLineToExample extends Application {
   @Override
   public void start(Stage stage) {

      //Creating an object of the Path class
      Path path = new Path();

      //Moving to the starting point
      MoveTo moveTo = new MoveTo();
      moveTo.setX(100.0);
      moveTo.setY(150.0);

      //Instantiating the HLineTo class
      HLineTo hLineTo = new HLineTo();

      //Setting the properties of the path element hroizontal line
      hLineTo.setX(10.0);

      //Adding the path elements to Observable list of the Path class
      path.getElements().add(moveTo);
      path.getElements().add(hLineTo);

      //Creating a Group object
      Group root = new Group(path);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Drawing a horizontal line");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
```

```
      stage.show();

   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac HLineToExample.java
java HLineToExample
```

On executing, the above program generates a JavaFX window displaying a horizontal line, which is drawn from the current position to the specified point, as shown below.



PathElement – Vertical Line

The path element **Vertical Line** is used to draw a vertical line to a point in the specified coordinates from the current position.

It is represented by a class named **VLineTo.** This class belongs to the package **javafx.scene.shape**.

This class has a property of the double datatype namely –

- **Y:** The y coordinate of the point to which a vertical is to be drawn from the current position.

To draw the path element vertical line, you need to pass a value to this property. This can be done either by passing it to the constructor of this class at the time of instantiation as follows —

```
LineTO line = new LineTo(x);
```

Or, by using its respective setter methods as follows —

```
setY(value);
```

To draw a vertical line to a specified point from the current position in JavaFX, follow the steps given below.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application** and implement the **start()** method of this class as follows.

```
public class ClassName extends Application {


    @Override
    public void start(Stage primaryStage) throws Exception {


    }


}
```

Step 2: Create the path class object

Take a look at the following code block to create the path class object.

```
//Creating a Path object
Path path = new Path();
```

Step 3: Create a Path

Create the **MoveTo** path element and set XY coordinates to the starting point of the line to the coordinates (100, 150). This can be done by using the methods **setX()** and **setY()** of the class **MoveTo** as shown below.

```
//Moving to the starting point
MoveTo moveTo = new MoveTo();
moveTo.setX(100.0f);
moveTo.setY(150.0f);
```

Step 4: Creating an object of the class VLineTo

Create the path element vertical line by instantiating the class named **VLineTo,** which belongs to the package **javafx.scene.shape** as follows.

```
//Creating an object of the class VLineTo
VLineTo vLineTo = new VLineTo();
```

Step 5: Setting Properties to the Element Vertical Line

Specify the coordinates of the point to which a vertical line is to be drawn from the current position. This can be done by setting the properties x and y using their respective setter methods as shown in the following code block.

```
//Setting the Properties of the vertical line element
lineTo.setX(500.0f);
lineTo.setY(150.0f);
```

Step 6: Adding Elements to the Observable List of the Path Class

Add the path elements **MoveTo** and **VlineTo** created in the previous steps to the observable list of the **Path** class as follows —

```
//Adding the path elements to Observable list of the Path class
path.getElements().add(moveTo);
path.getElements().add(VlineTo);
```

Step 7: Creating a Group Object

Create a group object by instantiating the class named Group, which belongs to the package **javafx.scene**.

Pass the Line (node) object created in the previous step as a parameter to the constructor of the Group class. This can be done in order to add it to the group as shown below –

```
Group root = new Group(line);
```

Step 8: Creating a Scene Object

Create a Scene by instantiating the class named **Scene** which belongs to the package **javafx.scene.** To this class, pass the Group object (**root**) created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen along with the object of the Group class as follows.

```
Scene scene = new Scene(group, 600, 300);
```

Step 9: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object, which is passed to the start method of the scene class as a parameter.

Using the **primaryStage** object, set the title of the scene as a **Sample Application** as shown below.

```
primaryStage.setTitle("Sample Application");
```

Step 10: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using this method as follows.

```
primaryStage.setScene(scene);
```

Step 11: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 12: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as shown below.

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

Example

Following is a program which draws a horizontal line from the current point to a specified position using the class **Path** of JavaFX. Save this code in a file with the name **HLineToExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.shape.HLineTo;
import javafx.scene.shape.MoveTo;
import javafx.scene.shape.Path;

public class HLineToExample extends Application {
   @Override

   public void start(Stage stage) {

      //Creating an object of the Path class
      Path path = new Path();

      //Moving to the starting point
      MoveTo moveTo = new MoveTo();

      moveTo.setX(100.0);
      moveTo.setY(150.0);

      //Instantiating the HLineTo class
      HLineTo hLineTo = new HLineTo();

      //Setting the properties of the path element hroizontal line
      hLineTo.setX(10.0);

      //Adding the path elements to Observable list of the Path class
      path.getElements().add(moveTo);
      path.getElements().add(hLineTo);

      //Creating a Group object
      Group root = new Group(path);

      //Creating a scene object
```

```
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Drawing a horizontal line");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();


   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac HLineToExample.java
java HLineToExample
```

On executing, the above program generates a JavaFX Window displaying a horizontal line, which is drawn from the current position to the specified point, as shown below.

PathElement – Quadratic Curve

The path element quadratic curve is used to draw a **quadratic curve** to a point in the specified coordinates from the current position.

It is represented by a class named **QuadraticCurveTo.** This class belongs to the package **javafx.scene.shape**.

This class has 4 properties of the double datatype namely –

- **setX:** The x coordinate of the point to which a curve is to be drawn from the current position.
- **setX:** The y coordinate of the point to which a curve is to be drawn from the current position.
- **controlX:** The x coordinate of the control point of the curve.
- **controlY:** The y coordinate of the control point of the curve.

To draw a quadratic curve, you need to pass values to these properties. This can be done either by passing them to the constructor of this class, in the same order, at the time of instantiation, as follows –

```
QuadCurveTo quadcurve = new QuadCurveTo(X, Y, controlX, controlY);
```

Or, by using their respective setter methods as shown below –

```
setX(value);
setY(value);
setControlX(value);
setControlY(value);
```

To draw a quadratic curve to a specified point from the current position in JavaFX, follow the steps given below.

Step 1: Creating a Class

Create a Java class and inherit the Application class of the package **javafx.application** and implement the **start()** method of this class as follows.

```
public class ClassName extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {

    }
}
```

Step 2: Create the Path Class Object

Create the Path Class Object as shown below.

```
//Creating a Path object
Path path = new Path();
```

Step 3: Create a Path

Create the **MoveTo** path element and set the XY coordinates to the starting point of the line to the coordinates (100, 150). This can be done by using the methods **setX()** and **setY()** of the class **MoveTo** as shown below.

```
//Moving to the starting point
MoveTo moveTo = new MoveTo();
moveTo.setX(100.0f);
moveTo.setY(150.0f);
```

Step 4: Creating an Object of the Class QuadCurveTo

Create the path element Quadratic Curve by instantiating the class named **QuadCurveTo** which belongs to the package **javafx.scene.shape** as follows.

```
//Creating an object of the class QuadCurveTo
QuadCurveTo quadCurveTo= new QuadCurveTo();
```

Step 5: Setting Properties to the Quadratic Curve Element

Specify the coordinates of the point to which a Quadratic Curve is to be drawn from the current position. Then you should set the properties x, y, controlx, controlY and the coordinates of the control point by their setter methods as shown below.

```
//Setting properties of the class QuadCurve
quadCurveTo.setX(500.0f);
quadCurveTo.setY(220.0f);
quadCurveTo.setControlX(250.0f);
quadCurveTo.setControlY(0.0f);
```

Step 6: Adding Elements to the Observable List of the Path Class

Add the path elements **MoveTo** and **QuadraticCurveTo** created in the previous steps to the observable list of the **Path** class as follows —

```
//Adding the path elements to Observable list of the Path class
path.getElements().add(moveTo);
path.getElements().add(quadCurveTo);
```

Step 7: Creating a Group Object

Create a group object by instantiating the class named Group, which belongs to the **package javafx.scene**.

Pass the Line (node) object created in the previous step as a parameter to the constructor of the Group class. This can be done in order to add it to the group as shown below –

```
Group root = new Group(line);
```

Step 8: Creating a Scene Object

Create a Scene by instantiating the class named **Scene** which belongs to the package **javafx.scene.** To this class, pass the Group object (**root**) created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 9: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object, which is passed to the start method of the scene class as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows:

```
primaryStage.setTitle("Sample Application");
```

Step 10: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using this method as shown below.

```
primaryStage.setScene(scene);
```

Step 11: Displaying the contents of the stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 12: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows:

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

Example

Following is a program which draws a quadratic curve from the current point to a specified position using the class named **Path** of JavaFX. Save this code in a file with the name **QuadCurveToExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.shape.MoveTo;
import javafx.scene.shape.Path;
```

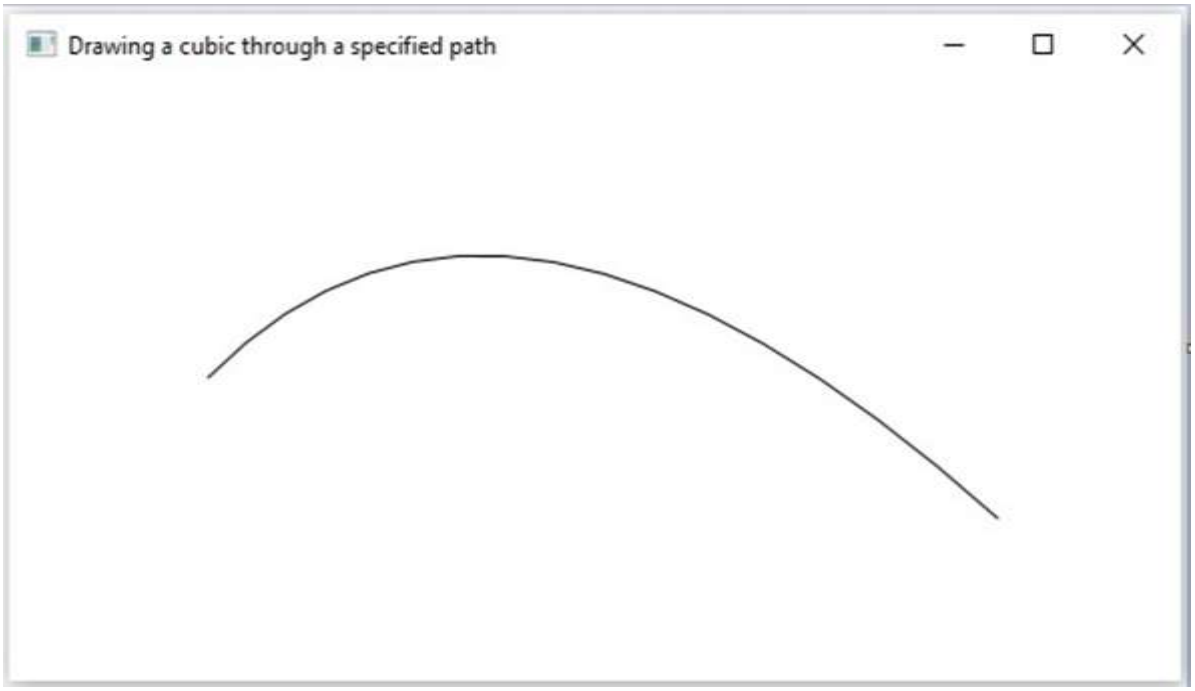
```
import javafx.scene.shape.QuadCurveTo;

public class QuadCurveToExample extends Application {
   @Override
   public void start(Stage stage) {

      //Creating an object of the class named Path
      Path path = new Path();

      //Moving to the starting point
      MoveTo moveTo = new MoveTo();
      moveTo.setX(100.0);
      moveTo.setY(150.0);

      //Instantiating the class QuadCurve
      QuadCurveTo quadCurveTo = new QuadCurveTo();

      //Setting properties of the class QuadCurve
      quadCurveTo.setX(500.0f);
      quadCurveTo.setY(220.0f);
      quadCurveTo.setControlX(250.0f);

      quadCurveTo.setControlY(0.0f);

      //Adding the path elements to Observable list of the Path class
      path.getElements().add(moveTo);
      path.getElements().add(quadCurveTo);
      //Creating a Group object
      Group root = new Group(path);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Drawing a cubic through a specified path");
```

```
        //Adding scene to the satge
        stage.setScene(scene);

        //Displaying the contents of the stage
        stage.show();

        }

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

    }

}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac QuadCurveToExample.java
java QuadCurveToExample
```

On executing, the above program generates a JavaFX window displaying a quadratic curve. This is drawn from the current position to the specified point as shown below.



PathElement – Cubic Curve

The path element cubic curve is used to draw a **cubic curve** to a point in the specified coordinates from the current position.

It is represented by a class named **CubicCurveTo.** This class belongs to the package **javafx.scene.shape**.

This class has 6 properties of the double datatype namely –

- **setX:** The x coordinate of the point to which a curve is to be drawn from the current position.
- **setX:** The y coordinate of the point to which a curve is to be drawn from the current position.
- **controlX1:** The x coordinate of the 1st control point of the curve.
- **controlY1:** The y coordinate of the 1st control point of the curve.
- **controlX2:** The x coordinate of the 2nd control point of the curve.
- **controlY2:** The y coordinate of the 2nd control point of the curve.

To draw a cubic curve, you need to pass values to these properties. This can be done by passing them to the constructor of this class. These should be in the same order as they were at the time of instantiation as shown below –

```
CubicCurveTo cubiccurve = new CubicCurveTo(X, Y, controlX1, controlY1,
controlX2, controlY2);
```

Or, by using their respective setter methods as follows –

```
setX(value);
setY(value);
setControlX1(value);
setControlY1(value);
setControlX2(value);
setControlY2(value);
```

To draw a cubic curve to a specified point from the current position in JavaFX, follow the steps given below.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application**. Then you can implement the **start()** method of this class as shown below.

```
public class ClassName extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
    }
}
```

Step 2: Create the Path Class Object

Create the path class object as shown in the following code block.

```
//Creating a Path object
Path path = new Path();
```

Step 3: Create a Path

Create the **MoveTo** path element and set the XY coordinates to the starting point of the line to the coordinates (100, 150). This can be done using the methods **setX()** and **setY()** of the class **MoveTo** as shown below.

```
//Moving to the starting point
MoveTo moveTo = new MoveTo();
moveTo.setX(100.0f);
moveTo.setY(150.0f);
```

Step 4: Creating an Object of the Class CubicCurveTo

Create the path element quadratic curve by instantiating the class named **CubicCurveTo**, which belongs to the package **javafx.scene.shape** as shown below –

```
//Creating an object of the class CubicCurveTo
CubicCurveTo cubicCurveTo= new CubicCurveTo ();
```

Step 5: Setting Properties to the Cubic Curve Element

Specify the coordinates of the point to which a cubic curve is to be drawn from the current position. Then you should set the properties x, y, controlX1, controlY1, controlX2, controlY2 and the coordinates of the control point by their setter methods as shown below.

```
//Setting properties of the class CubicCurve
cubicCurveTo.setControlX1(400.0f);
cubicCurveTo.setControlY1(40.0f);
cubicCurveTo.setControlX2(175.0f);
cubicCurveTo.setControlY2(250.0f);
```

```
cubicCurveTo.setX(500.0f);
cubicCurveTo.setY(150.0f)
```

Step 6: Adding Elements to the Observable List of Path Class

Add the path elements → **MoveTo** and **CubicCurveTo**, created in the previous steps to the observable list of the **Path** class as follows —

```
//Adding the path elements to Observable list of the Path class
path.getElements().add(moveTo);
path.getElements().add(cubicCurveTo);
```

Step 7: Creating a Group Object

Create a group object by instantiating the class named **Group**, which belongs to the package **javafx.scene**.

Pass the Line (node) object created in the previous step as a parameter to the constructor of the Group class. This can be done in order to add it to the group as shown below –

```
Group root = new Group(line);
```

Step 8: Creating a Scene Object

Create a scene by instantiating the class named **Scene,** which belongs to the package **javafx.scene.** To this class, pass the Group object (**root**), created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 9: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is the Stage object which is passed to the start method of the scene class, as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 10: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using this method as follows.

```
primaryStage.setScene(scene);
```

Step 11: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 12: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows –

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

Example

Following is the program which draws a cubic curve from the current point to a specified position using the class named **Path** of JavaFX. Save this code in a file with the name **CubicCurveToExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.shape.CubicCurveTo;
import javafx.scene.shape.MoveTo;
import javafx.scene.shape.Path;

public class CubicCurveToExample extends Application {

   @Override
   public void start(Stage stage) {

      //Creating an object of the class named Path
      Path path = new Path();

      //Moving to the starting point
      MoveTo moveTo = new MoveTo();
      moveTo.setX(100.0);
      moveTo.setY(150.0);
```

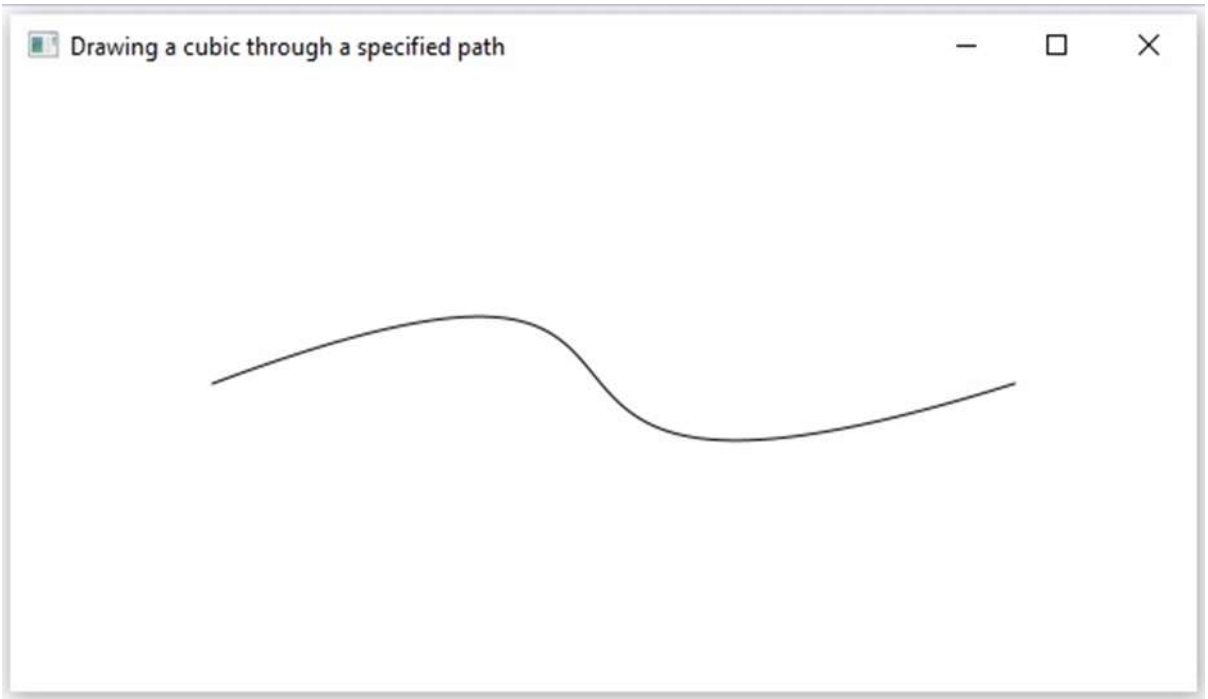
```
      //Instantiating the class CubicCurve
      CubicCurveTo cubicCurveTo = new CubicCurveTo();

      //Setting properties of the class CubicCurve
      cubicCurveTo.setControlX1(400.0f);
      cubicCurveTo.setControlY1(40.0f);
      cubicCurveTo.setControlX2(175.0f);
      cubicCurveTo.setControlY2(250.0f);
      cubicCurveTo.setX(500.0f);
      cubicCurveTo.setY(150.0f);

      //Adding the path elements to Observable list of the Path class
      path.getElements().add(moveTo);
      path.getElements().add(cubicCurveTo);
      //Creating a Group object
      Group root = new Group(path);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Drawing a cubic through a specified path");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

      }

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

   }
```

```
}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac cubicCurveToExample.java
java cubicCurveToExample
```

On executing, the above program generates a javaFx window displaying a quadratic curve, which is drawn from the current position to the specified point, as shown below.



PathElement – Arc

The Path Element **Arc** is used to draw an arc to a point in the specified coordinates from the current position.

It is represented by a class named **ArcTo.** This class belongs to the package **javafx.scene.shape**.

This class has 4 properties of the double datatype namely –

- **X:** The x coordinate of the center of the arc.
- **Y:** The y coordinate of the center of the arc.
- **radiusX:** The width of the full ellipse of which the current arc is a part of.
- **radiusY:** The height of the full ellipse of which the current arc is a part of.

To draw the Path element arc, you need to pass values to these properties This can be done by passing them to the constructor of this class, in the same order, at the time of instantiation as follows –

```
ArcTo arcTo = new ArcTo(x, y, radius, radiusY);
```

Or, by using their respective setter methods as follows –

```
setX(value);
setY(value);
setRadiusX(value);
setRadiusY(value);
```

To draw an arc to a specified point from the current position in JavaFX, follow the steps given below.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application**. You can then implement the **start()** method of this class as follows.

```
public class ClassName extends Application {


    @Override
    public void start(Stage primaryStage) throws Exception {
    }
}
```

Step 2: Create the Path Class Object

Create the path class object as shown in the following code block.

```
//Creating a Path object
Path path = new Path();
```

Step 3: Create the Path

Create the **MoveTo** path element and set XY coordinates to starting point of the line to the coordinates (100, 150). This can be done by using the methods **setX()** and **setY()** of the class **MoveTo** as shown below.

```
//Moving to the starting point
MoveTo moveTo = new MoveTo();
moveTo.setX(100.0f);
moveTo.setY(150.0f);
```

Step 4: Creating an Object of the Class ArcTo

Create the path element arc by instantiating the class named **ArcTo**, which belongs to the package **javafx.scene.shape** as follows.

```
//Creating an object of the class ArcTo
ArcTo arcTo = new ArcTo();
```

Step 5: Setting Properties to the Arc Element

Specify the x, y coordinates of the center of the ellipse (of which this arc is a part of). Then you can specify the radiusX, radiusY, start angle, and length of the arc using their respective setter methods as shown below.

```
//setting properties of the path element arc
arcTo.setX(300.0);
arcTo.setY(50.0);

arcTo.setRadiusX(50.0);
arcTo.setRadiusY(50.0);
```

Step 6: Adding Elements to the Observable List of the Path Class

Add the path elements **moveTo** and **arcTo** created in the previous steps to the observable list of the **Path** class as follows —

```
//Adding the path elements to Observable list of the Path class
path.getElements().add(moveTo);
path.getElements().add(arcto);
```

Step 7: Creating a Group Object

Create a group object by instantiating the class named **Group**, which belongs to the package **javafx.scene**.

Pass the Line (node) object, created in the previous step as a parameter to the constructor of the Group class. This should be done in order to add it to the group as follows –

```
Group root = new Group(line);
```

Step 8: Creating a Scene Object

Create a Scene by instantiating the class named **Scene**, which belongs to the package **javafx.scene.** To this class, pass the Group object (**root**) created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 9: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object which is passed to the start method of the scene class, as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 10: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using the method given below.

```
primaryStage.setScene(scene);
```

Step 11: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 12: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows.

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

Example

Following is a program that draws an arc from the current point to a specified position using the class Path of JavaFX. Save this code in a file with the name **ArcToExample.java**.

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.shape.ArcTo;
```

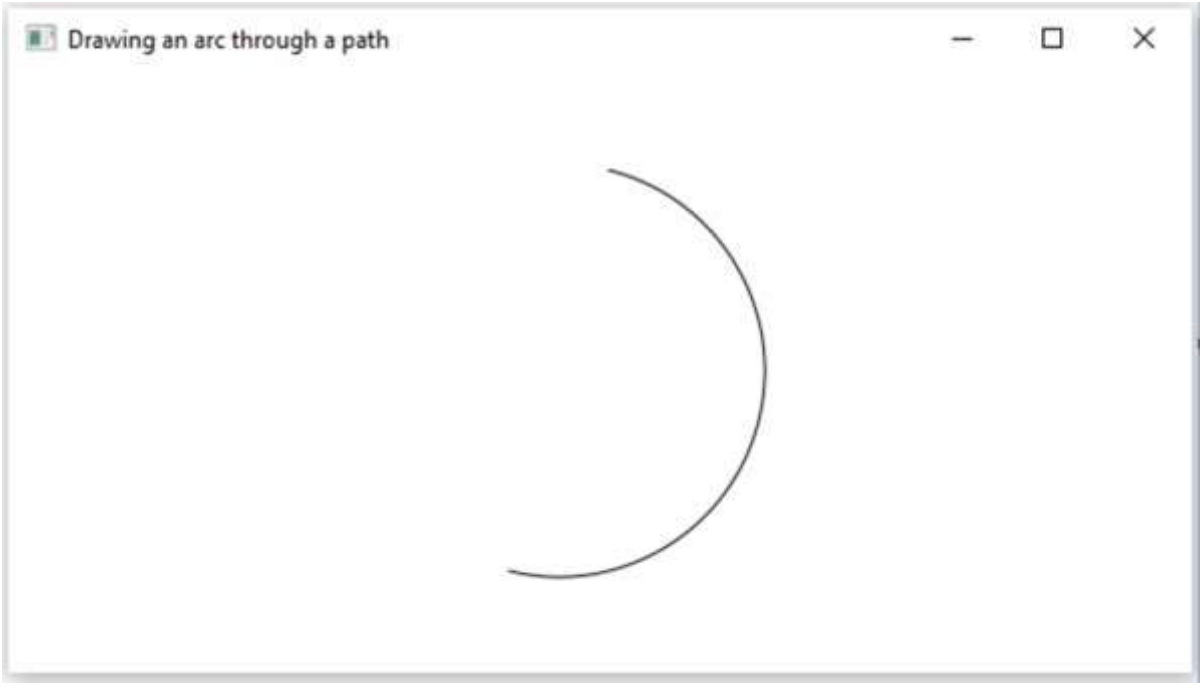
```
import javafx.scene.shape.MoveTo;
import javafx.scene.shape.Path;

public class ArcToExample extends Application {
   @Override
   public void start(Stage stage) {

      //Creating an object of the class Path
        Path path = new Path();

      //Moving to the starting point
      MoveTo moveTo = new MoveTo();
      moveTo.setX(250.0);
      moveTo.setY(250.0);

      //Instantiating the arcTo class
      ArcTo arcTo = new ArcTo();

      //setting properties of the path element arc
      arcTo.setX(300.0);
      arcTo.setY(50.0);

      arcTo.setRadiusX(50.0);
      arcTo.setRadiusY(50.0);

      //Adding the path elements to Observable list of the Path class
      path.getElements().add(moveTo);
      path.getElements().add(arcTo);

      //Creating a Group object
      Group root = new Group(path);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
```

```
      stage.setTitle("Drawing an arc through a path");

      //Adding scene to the stage
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();


   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac ArcToExample.java
java ArcToExample
```

On executing, the above program generates a JavaFX window displaying an arc, which is drawn from the current position to the specified point, as shown below.



Properties of 2D Objects

For all the 2-Dimensional objects, you can set various properties like fill, stroke, StrokeType, etc. The following section discusses various properties of 2D objects.

Stroke Type

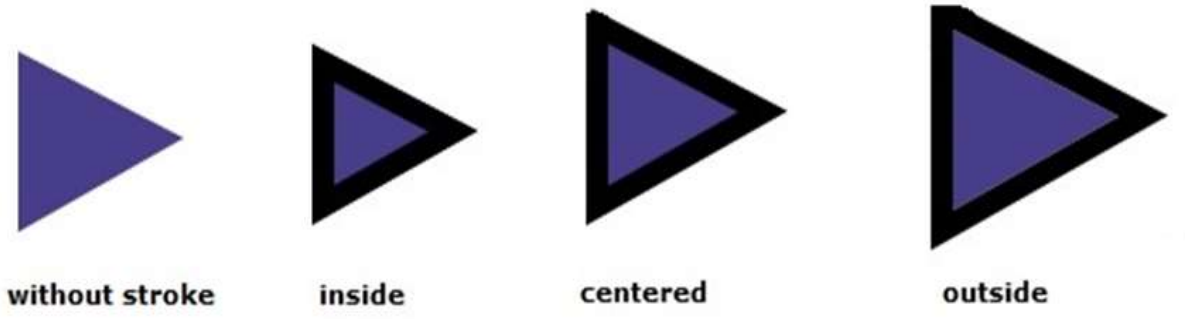
This property is of the type StrokeType. It represents the position of the boundary line applied to the shape. You can set the type of the stroke using the method **setStrokeType()** as follows –

```
Path.setStrokeType(StrokeType.CENTERED);
```

The stroke type of a shape can be –

- **Inside:** The boundary line will be drawn inside the edge (outline) of the shape (StrokeType.INSIDE).
- **Outside:** The boundary line will be drawn outside the edge (outline) of the shape (StrokeType.OUTSIDE).
- **Centered:** The boundary line will be drawn in such a way that the edge (outline) of the shape passes exactly thorough the center of the line (StrokeType.CENTERED).

By default, the stroke type of a shape is centered. Following is the diagram of a triangle with different Stroke Types:

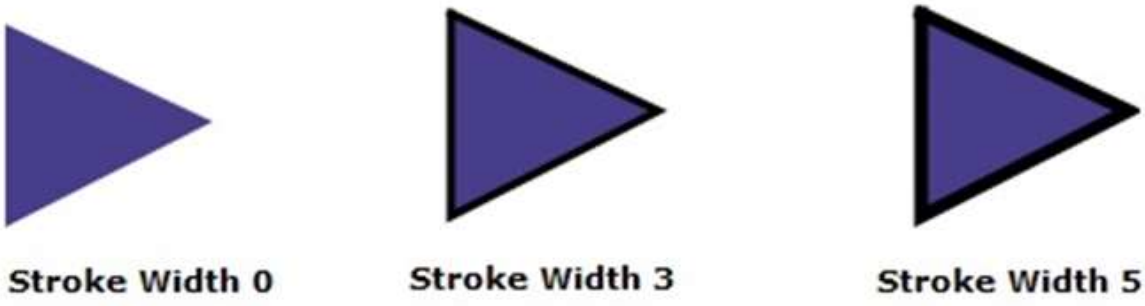


Stroke Width

This property is of the type double and it represents the width of the boundary line of the shape. You can set the stroke width using the method **setStrokeWidth()** as follows **–**

```
Path.setStrokeWidth(3.0)
```

By default, the value of the stroke with of a shape is **1.0**. Following is a diagram of a triangle with different values of stroke width.

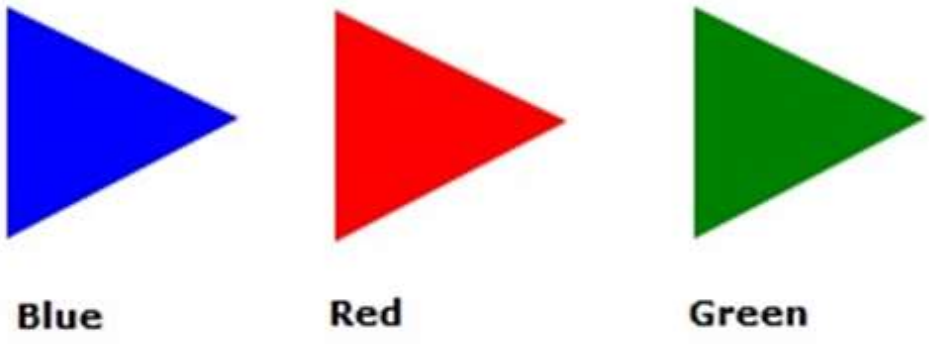


Stroke Fill

This property is of the type **Paint** and it represents the color that is to be filled inside the shape. You can set the fill color of a shape using the method **setFill()** as follows –

```
path.setFill(COLOR.BLUE);
```

By default, the value of the stroke color is **BLACK.** Following is a diagram of a triangle with different colors.



Stroke

This property is of the type **Paint** and it represents the color of the boundary line of the shape. You can set a value to this property using the method **setStroke()** as shown below –

```
path.setStroke(Color.RED);
```

By default, the color of the stroke is black. Following is a diagram of a triangle with different stroke colors.

Stroke Line Join

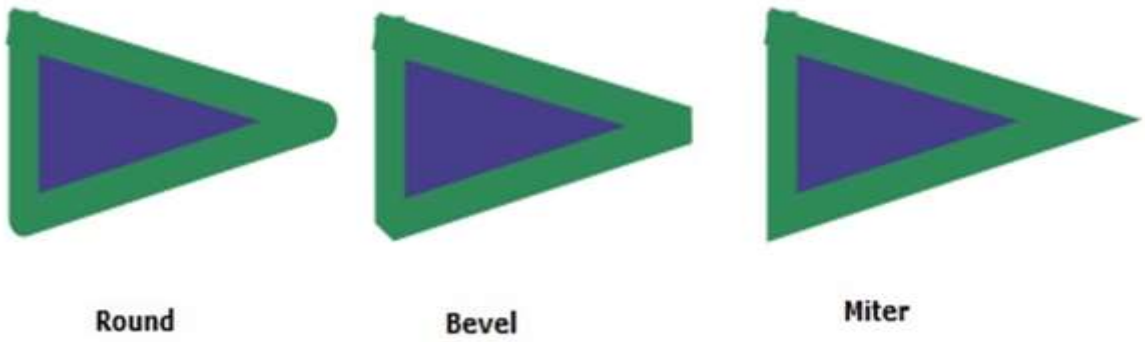
This property is of the type **StrokeLineJoin**, it represents the type of joining that is used at the edges of the shape. You can set the line join of the stroke using the method **setStrokeLineJoin()** as follows –

```
path.setStrokeLineJoin(StrokeLineJoin.BEVEL);
```

The stroke line join can be –

- **Bevel:** The bevel join is applied to the joining of the edges of the shape (StrokeLineJoin.BEVEL).
- **Miter:** The miter join is applied to the joining of the edges of the shape (StrokeLineJoin.MITER).
- **Round:** The round join is applied to the joining of the edges of the shape (StrokeLineJoin.ROUND).

By default, the Stroke Line Joining a shape is miter. Following is a diagram of a triangle with different line join types:



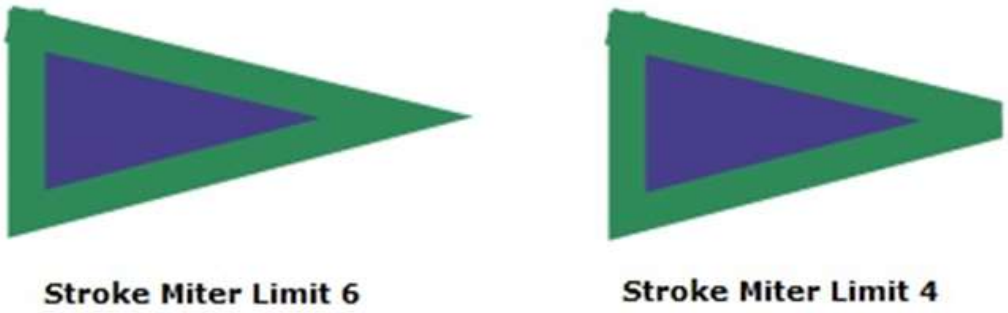
Stroke Miter Limit

This property is of the type double. It represents the limit for the distance between the inside point of the joint and the outside point of the joint. If the distance between these two points exceeds the given limit, the miter is cut at the edge.

You can set value to this property using the method **setStroke()** as follows –

```
path.setStrokeMiterLimit(4);
```

By default, the stroke miter limit value id 10 of the stroke is black. Following is a diagram of a triangle with different stroke limits.



Stroke Line Cap

This property is of the type **StrokeLineCap** It represents the end cap style of the line. You can set the line cap stroke using the method **setStrokeLineCap()** as shown in the following code block –

```
line.setStrokeLineCap(StrokeLineCap.SQUARE);
```

The stroke line cap can be –

- **Butt:** The butt line cap is applied at the end of the lines (StrokeLineCap.BUTT).
- **Square:** The square line cap is applied at the end of the lines (StrokeLineCap.SQUARE).
- **Round:** The round line cap is applied at the end of the lines (StrokeLineCap.ROUND).

By default, the Stroke Line cap a shape is square. Following is the diagram of a triangle with different line cap types.

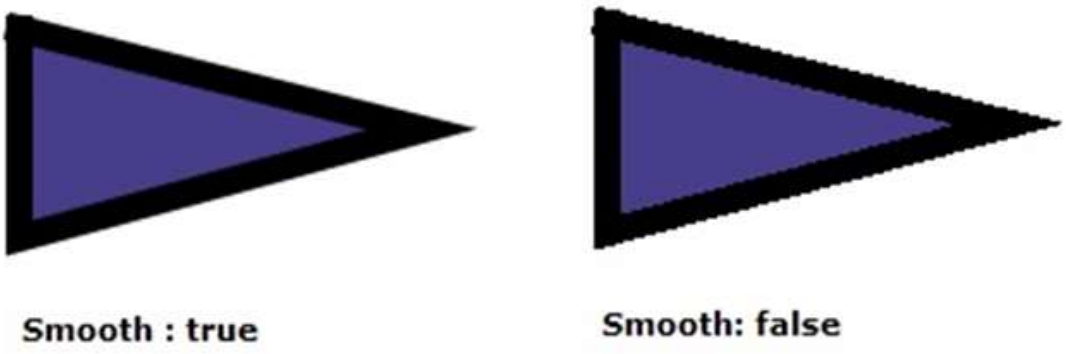
Smooth

This property is of the type Boolean. If this value is true, then the edges of the shape will be smooth.

You can set value to this property using the method **setSmooth()** as follows –

```
path.setSmooth(false);
```

By default, the smooth value is true. Following is a diagram of a triangle with both smooth values.



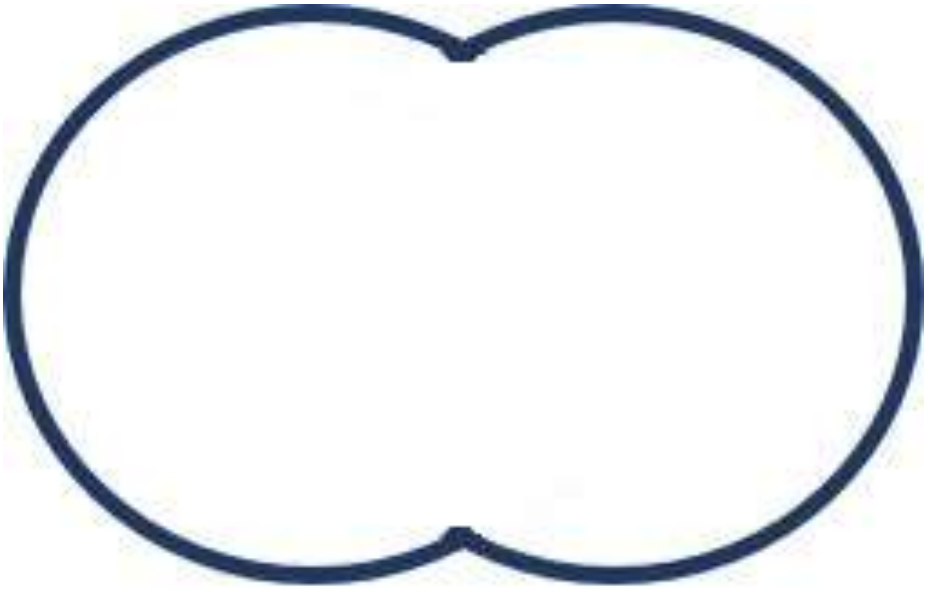
Operations on 2D Objects

If we add more than one shape to a group, the first shape is overlapped by the second one as shown below.

In addition to the transformations (rotate, scale, translate, etc.), transitions (animations), you can also perform three operations on 2D objects namely – **Union**, **Subtraction** and **Intersection**.

Union Operation

This operation takes two or more shapes as inputs and returns the area occupied by them combinedly as shown below.



Union Operation

You can perform union operation on the shapes using the method called **union().** Since this is a static method, you should call it using the class name (Shape or its subclasses) as shown below.

```
Shape shape = Shape.subtract(circle1, circle2);
```

Example

Following is an example of the union operation. In here, we are drawing two circles and performing a union operation on them. Save this code in a file with the name **unionExample.java.**

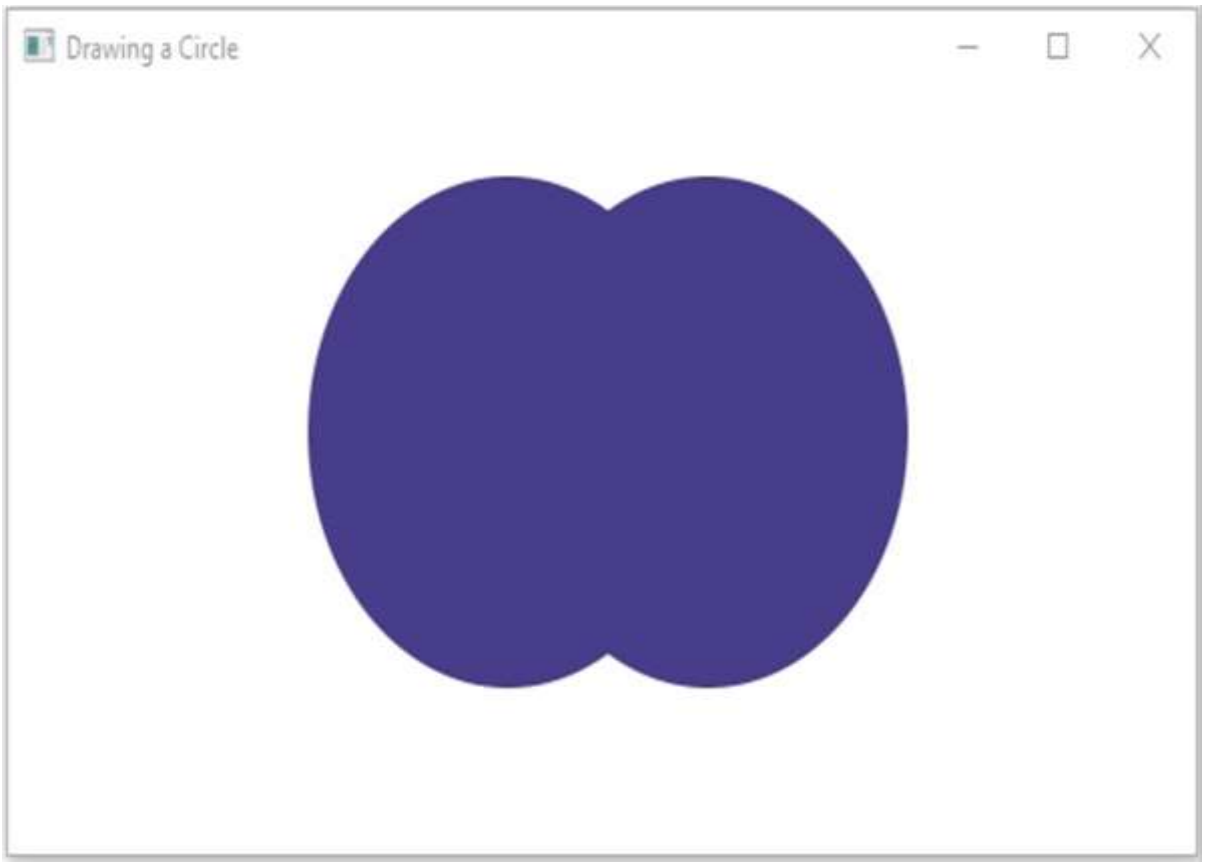
```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Shape;

public class UnionExample extends Application {

   @Override
   public void start(Stage stage) {

      //Drawing Circle1
      Circle circle1 = new Circle();
      //Setting the position of the circle
      circle1.setCenterX(250.0f);
      circle1.setCenterY(135.0f);
      //Setting the radius of the circle
      circle1.setRadius(100.0f);
      //Setting the color of the circle
      circle1.setFill(Color.DARKSLATEBLUE);

      //Drawing Circle2
      Circle circle2 = new Circle();
      //Setting the position of the circle
      circle2.setCenterX(350.0f);
      circle2.setCenterY(135.0f);
```

```
      //Setting the radius of the circle
      circle2.setRadius(100.0f);
      //Setting the color of the circle

      circle2.setFill(Color.BLUE);

      //Performing union operation on the circle
      Shape shape = Shape.union(circle1, circle2);
      //Setting the fill color to the result
      shape.setFill(Color.DARKSLATEBLUE);

      //Creating a Group object
      Group root = new Group(shape);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Union Example");

      //Adding scene to the stage

      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }

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

Compile and execute the saved java file from the command prompt using the following commands.

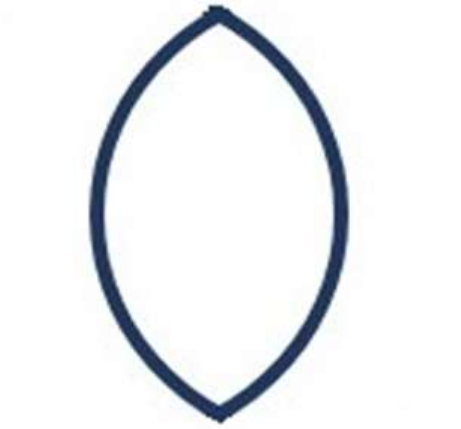
```
javac UnionExample.java
java UnionExample
```

On executing, the above program generates a JavaFX window displaying the following output –



Intersection Operation

This operation takes two or more shapes as inputs and returns the intersection area between them as shown below.



Intersection operation

You can perform an intersection operation on the shapes using the method named **intersect().** Since this is a static method, you should call it using the class name (Shape or its subclasses) as shown below.

```
Shape shape = Shape.intersect(circle1, circle2);
```

Following is an example of the intersection operation. In here, we are drawing two circles and performing an intersection operation on them.

Save this code in a file with the name **IntersectionExample.java**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Shape;

public class IntersectionExample extends Application {
   @Override
   public void start(Stage stage) {

      //Drawing Circle1
      Circle circle1 = new Circle();
```

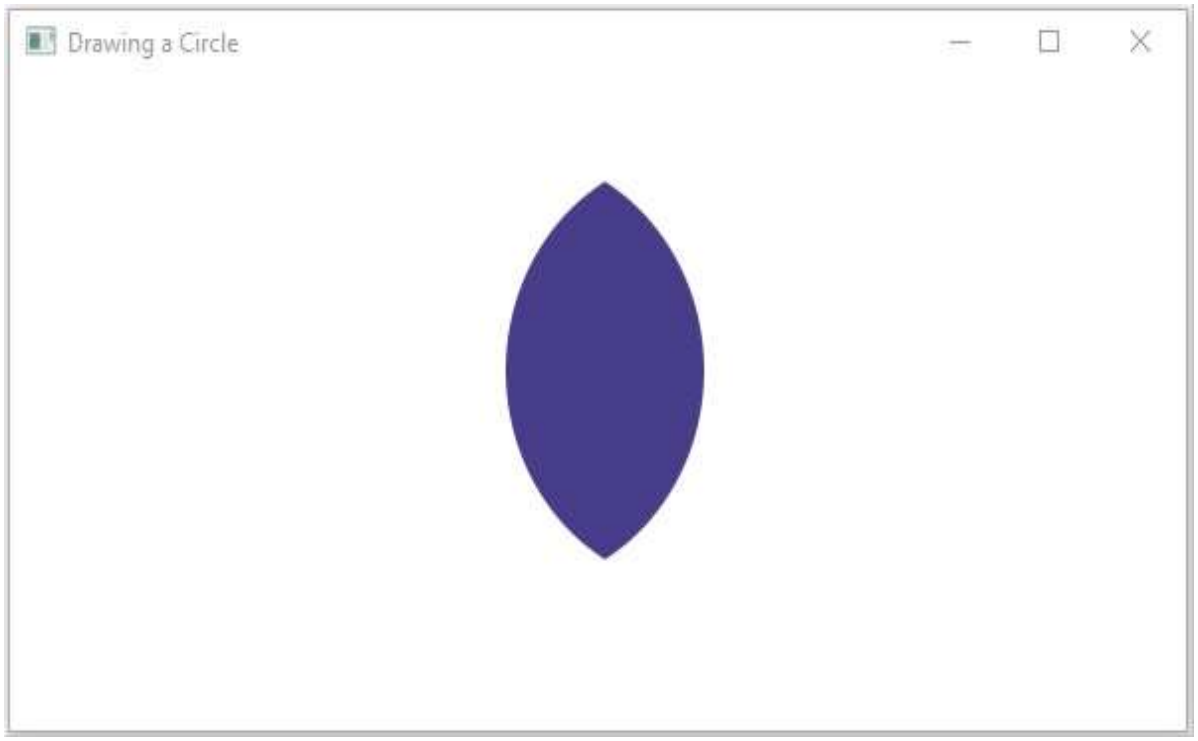
```
        //Setting the position of the circle
        circle1.setCenterX(250.0f);
        circle1.setCenterY(135.0f);
        //Setting the radius of the circle
        circle1.setRadius(100.0f);
        //Setting the color of the circle
        circle1.setFill(Color.DARKSLATEBLUE);

        //Drawing Circle2
        Circle circle2 = new Circle();
        //Setting the position of the circle
        circle2.setCenterX(350.0f);
        circle2.setCenterY(135.0f);
        //Setting the radius of the circle

        circle2.setRadius(100.0f);
        //Setting the color of the circle
        circle2.setFill(Color.BLUE);

        //Performing intersection operation on the circle
        Shape shape = Shape.intersect(circle1, circle2);
        //Setting the fill color to the result
        shape.setFill(Color.DARKSLATEBLUE);

        //Creating a Group object
        Group root = new Group(shape);

        //Creating a scene object
        Scene scene = new Scene(root, 600, 300);

        //Setting title to the Stage
        stage.setTitle("Intersection Example");

        //Adding scene to the stage
        stage.setScene(scene);
```

```
      //Displaying the contents of the stage
      stage.show();

   }

   public static void main(String args[]){

      launch(args);

   }

}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac IntersectionExample.java
java IntersectionExample
```

On executing, the above program generates a JavaFX window displaying the following output –



Subtraction Operation

This operation takes two or more shapes as an input. Then, it returns the area of the first shape excluding the area overlapped by the second one as shown below.

Substraction Operation

You can perform the Substraction Operation on the shapes using the method named **substract().** Since this is a static method, you should call it using the class name (Shape or its subclasses) as shown below.

```
Shape shape = Shape.subtract(circle1, circle2);
```

Following is an example of the Subtraction Operation. In here, we are drawing two circles and performing a subtraction operation on them.

Save this code in a file with name **SubtractionExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Shape;

public class SubtractionExample extends Application {
   @Override
```

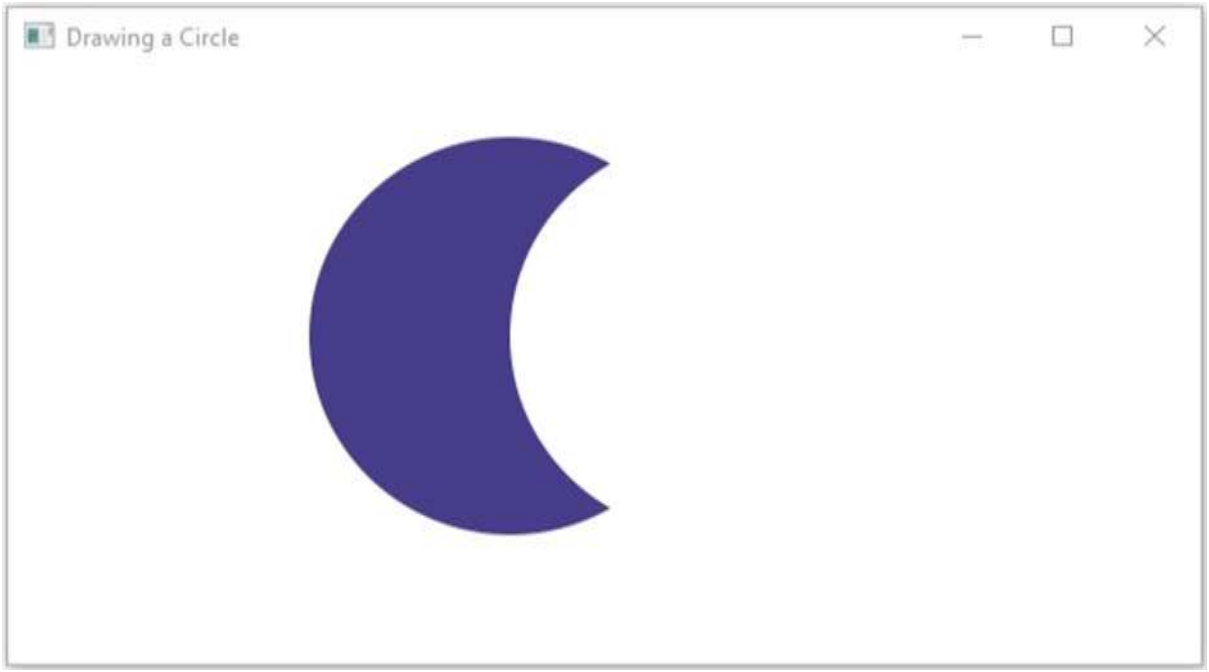
```
public void start(Stage stage) {

   //Drawing Circle1

  Circle circle1 = new Circle();
   //Setting the position of the circle
   circle1.setCenterX(250.0f);
   circle1.setCenterY(135.0f);
   //Setting the radius of the circle
   circle1.setRadius(100.0f);
   //Setting the color of the circle
   circle1.setFill(Color.DARKSLATEBLUE);

   //Drawing Circle2
   Circle circle2 = new Circle();
   //Setting the position of the circle
   circle2.setCenterX(350.0f);
   circle2.setCenterY(135.0f);
   //Setting the radius of the circle
   circle2.setRadius(100.0f);
   //Setting the color of the circle
   circle2.setFill(Color.BLUE);

   //Performing subtraction operation on the circle
   Shape shape = Shape.subtract(circle1, circle2);
   //Setting the fill color to the result
   shape.setFill(Color.DARKSLATEBLUE);

   //Creating a Group object
   Group root = new Group(shape);

   //Creating a scene object
   Scene scene = new Scene(root, 600, 300);

   //Setting title to the Stage
   stage.setTitle("Subtraction Example");
```

```
      //Adding scene to the stage
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac SubtractionExample.java
java SubtractionExample
```

On executing, the above program generates a JavaFX window displaying the following output –



6. JavaFX – Text

Just like various shapes, you can also create a text node in JavaFX. The text node is represented by the class named **Text**, which belongs to the package **javafx.scene.text**.

This class contains several properties to create text in JavaFX and modify its appearance. This class also inherits the Shape class which belongs to the package **javafx.scene.shape.**

Therefore, in addition to the properties of the text like font, alignment, line spacing, text, etc. It also inherits the basic shape node properties such as **strokeFill**, **stroke**, **strokeWidth**, **strokeType**, etc.

Creating a Text Node

Since the class Text of the package **javafx.scene.text** represents the text node in JavaFX, you can create a text by instantiating this class as follows –

```
Text text = new Text();
```

The class Text contains a property named **text** of string type, which represents the text that is to be created.

After instantiating the Text class, you need to set value to this property using the **setText()** method as shown below.

```
String text = "Hello how are you"
Text.setText(text);
```

You can also set the position (origin) of the text by specifying the values to the properties x and y using their respective setter methods namely **setX()** and **setY()** as shown in the following code block –

```
text.setX(50);
text.setY(50);
```

Example

The following program is an example demonstrating how to create a text node in JavaFX. Save this code in a file with name **TextExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
```

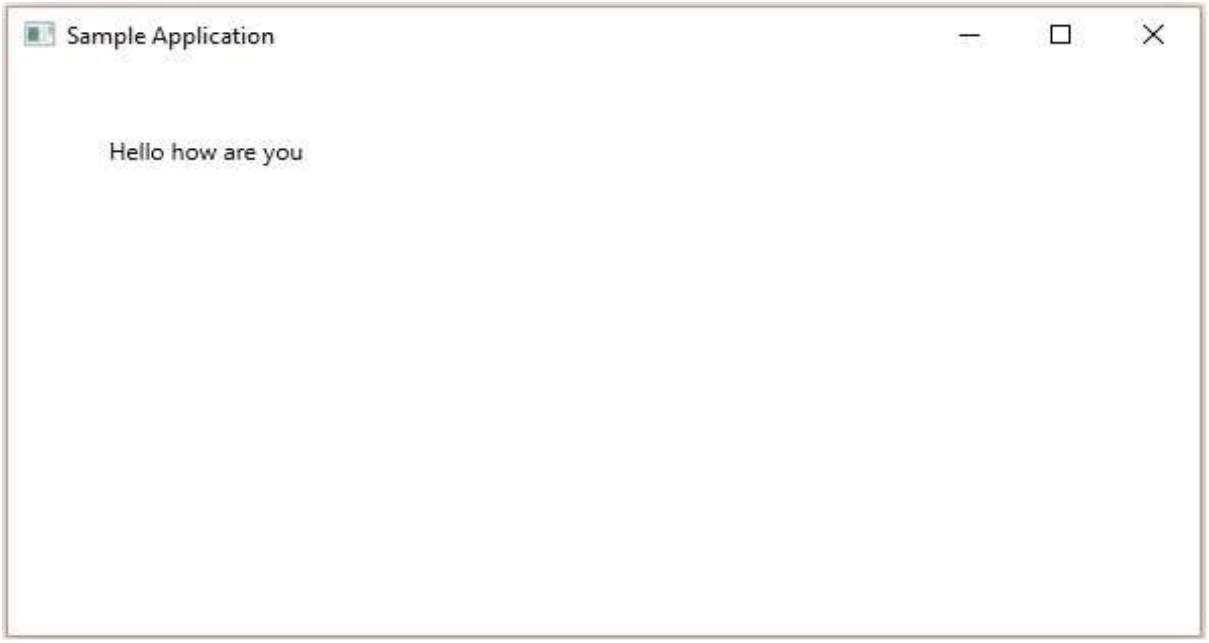
```
import javafx.stage.Stage;
import javafx.scene.text.Text;

public class TextExample extends Application {

   @Override
   public void start(Stage stage) {

      //Creating a Text object
      Text text = new Text();

      //Setting the text to be added.
      text.setText("Hello how are you");

      //setting the position of the text
      text.setX(50);
      text.setY(50);

      //Creating a Group object
      Group root = new Group(text);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Sample Application");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }
```

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac TextExample.java
java TextExample
```

On executing, the above program generates a JavaFX window displaying the specified text as follows –



Position and Font of the Text

By default, the text created by text class is of the font…, size…, and black in color.

You can change the font size and color of the text using the **setFont()** method. This method accepts an object of the **Font** class.

The class named **Font** of the package **javafx.scene.text** is used to define the font for the text. This class contains a static method named **font().**

This method accepts four parameters namely –

- **family:** This is of a String type and represents the family of the font that we want to apply to the text.

- **weight:** This property represents the weight of the font. It accepts 9 values, which are: ***FontWeight.BLACK***, ***FontWeight.BOLD***, ***FontWeight.EXTRA_BOLD***, ***FontWeight.EXTRA_LIGHT***, ***LIGHT***, ***MEDIUM***, ***NORMAL***, ***SEMI_BOLD***, ***THIN***.

- **posture:** This property represents the font posture (regular or italic). It accepts two values ***FontPosture.REGULAR*** and ***FontPosture.ITALIC***.

- **size:** This property is of type double and it represents the size of the font.

You can set font to the text by using the following method –

```
text.setFont(Font.font("verdana", FontWeight.BOLD, FontPosture.REGULAR, 20));
```

Example

The following program is an example demonstrating how to set font of the text node in JavaFX. In here, we are setting the font to Verdana, weight to bold, posture to regular and size to 20.

Save this code in a file with the name **TextFontExample.java.**

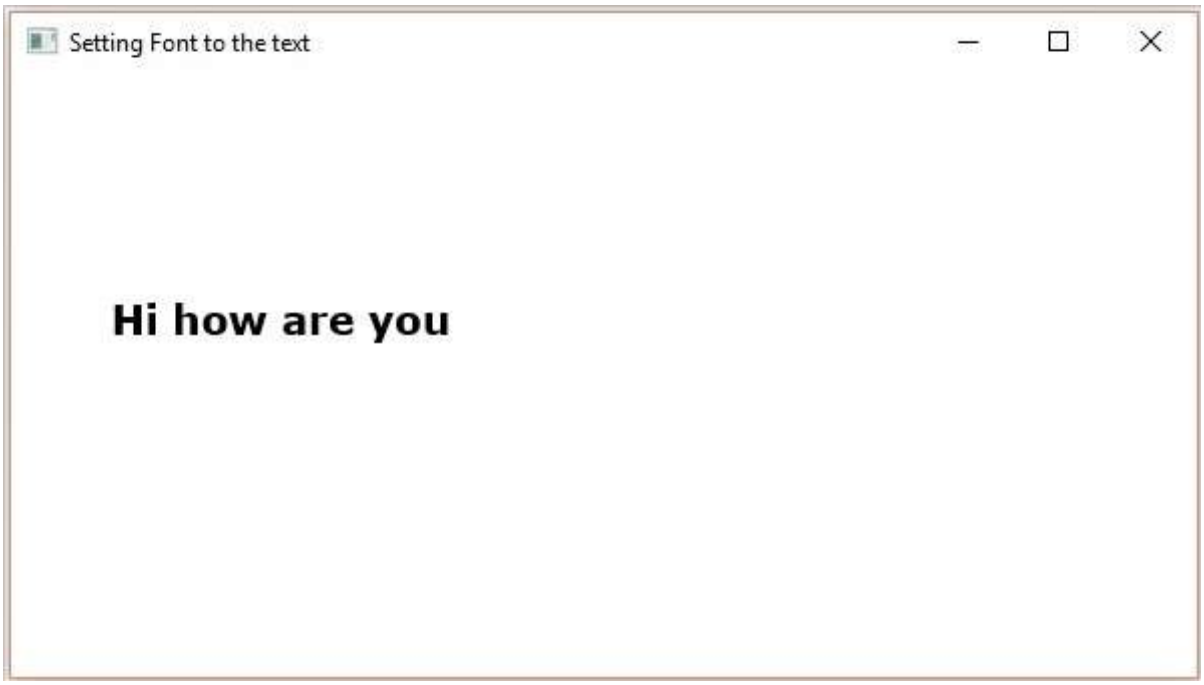
```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.text.Font;
import javafx.scene.text.FontPosture;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;

public class TextFontExample extends Application {
   @Override
   public void start(Stage stage) {

      //Creating a Text object
      Text text = new Text();



      //Setting font to the text
      text.setFont(Font.font("verdana", FontWeight.BOLD, FontPosture.REGULAR,
20));


      //setting the position of the text
```

```
      text.setX(50);
      text.setY(130);

      //Setting the text to be added.
      text.setText("Hi how are you");

      //Creating a Group object
      Group root = new Group(text);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Setting Font to the text");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac TextFontExample.java
java TextFontExample
```

On executing, the above program generates a JavaFX window displaying the text with the specified font as follows –



Stroke and Color

The Text class also inherits the class Shape of the package. Therefore, you can use **javafx.scene.shape** with which you can set the stroke and color to the text node too.

You can set the color to the text using the **setFill()** method of the shape (inherited) class as follows –

```
text.setFill(Color.BEIGE);
```

Similarly, you can set the stroke color of the text using the method **setStroke().** While the width of the stroke can be set using the method **setStrokeWidth()** as follows **–**

```
//Setting the color
text.setFill(Color.BROWN);



//Setting the Stroke
text.setStrokeWidth(2);

//Setting the stroke color
text.setStroke(Color.BLUE);
```

Example

The following program is an example that demonstrates how to set the color, strokeWidth and strokeColor, of the text node. In this code, we are setting stroke color to – blue, text color to – brown and the stroke width to – 2.

Save this code in a file with the name **StrokeExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.scene.text.Font;
import javafx.scene.text.FontPosture;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;

public class StrokeExample extends Application {
   @Override
   public void start(Stage stage) {

      //Creating a Text object
      Text text = new Text();

      //Setting font to the text
      text.setFont(Font.font("verdana", FontWeight.BOLD, FontPosture.REGULAR,
50));

      //setting the position of the text

      text.setX(50);
      text.setY(130);

      //Setting the color
      text.setFill(Color.BROWN);

      //Setting the Stroke
      text.setStrokeWidth(2);
```

```
      // Setting the stroke color
      text.setStroke(Color.BLUE);

      //Setting the text to be added.
      text.setText("Hi how are you");

      //Creating a Group object
      Group root = new Group(text);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Setting font to the text");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac StrokeExample.java
java StrokeExample
```

On executing, the above program generates a JavaFX window displaying the text with the specified stroke and color attributes as follows –

Applying Decorations to Text

You can also apply decorations such as strike through; in which case a line is passed through the text. You can underline a text using the methods of the **Text** class.

You can strike through the text using the method **setStrikethrough().** This accepts a Boolean value, pass the value **true** to this method to strike through the text as shown in the following code box –

```
//Striking through the text
text1.setStrikethrough(true);
```

In the same way, you can underline a text by passing the value **true** to the method **setUnderLine()** as follows –

```
//underlining the text
text2.setUnderline(true);
```

Example

The following program is an example demonstrating how to apply decorations such as **underline** or **strike through** to a text. Save this code in a file with the name **DecorationsExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.stage.Stage;
```

```
import javafx.scene.text.Font;
import javafx.scene.text.FontPosture;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;

public class DecorationsExample extends Application {

   @Override
   public void start(Stage stage) {

      //Creating a Text_Example object
      Text text1 = new Text("Hi how are you");
      //Setting font to the text
      text1.setFont(Font.font("verdana", FontWeight.BOLD, FontPosture.REGULAR,
20));
      //setting the position of the text
      text1.setX(50);
      text1.setY(75);
      //Striking through the text
      text1.setStrikethrough(true);

      //Creating a Text_Example object

      Text text2 = new Text("Welcome to Tutorialspoint");
      //Setting font to the text
      text2.setFont(Font.font("verdana", FontWeight.BOLD, FontPosture.REGULAR,
20));
      //setting the position of the text
      text2.setX(50);
      text2.setY(150);

      //underlining the text
      text2.setUnderline(true);



      //Creating a Group object
      Group root = new Group(text1, text2);
```

```
      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Decorations Example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }

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

Compile and execute the saved Java file from the command prompt using the following commands.

```
javac DecorationsExample.java
java DecorationsExample
```

On executing, the above program generates a JavaFX window as shown below –


Decorations Example
Hi how are you
Welcome to Tutorialspoint

7. JavaFX – Effects

An effect is any action that enhances the appearance of the graphics. In JavaFX, an effect is an algorithm that is applied on nodes to enhance their appearance visually. The **effect** property of the **Node** class is used to specify the effect.

In JavaFX, you can set various effects to a node such as **bloom**, **blur** and **glow**. Each of these effects are represented by a class and all these classes are available in a package named **javafx.scene.effect.**

Applying Effects to a Node

You can apply an effect to a node using the **setEffect()** method. To this method, you need to pass the object of the effect.

To apply an effect to a node, you need to –

- Create the node.
- Instantiate the respective class of the effect that is needed to be applied.
- Set the properties of the effect.
- Apply the effect to the node using the **setEffect()** method.

Creating the Nodes

First of all, create the nodes in a JavaFX application by instantiating their respective classes.

For example, if you want to apply glow effect to an image in your application. Firstly, you need to create an image node by instantiating the Image class and set its view as shown below.

```
//Creating an image
Image image = new
Image("http://www.tutorialspoint.com/green/images/logo.png");


//Setting the image view
ImageView imageView = new ImageView(image);
//Setting the position of the image
imageView.setX(100);
imageView.setY(70);


//setting the fit height and width of the image view
imageView.setFitHeight(200);
```

```
imageView.setFitWidth(400);
//Setting the preserve ratio of the image view
imageView.setPreserveRatio(true);
```

Instantiating the Respective Class

Instantiate the class representing the effect that is needed to be applied to the created node.

For example: To apply the glow effect, you need to instantiate the **Glow** class as shown in the following code box –

```
Glow glow = new Glow();
```

Setting the Properties of the Effect

After instantiating the class, you need to set the properties for the effect using its setter methods.

For example: To draw a 3-Dimensional box, you need to pass its width, height and depth. You can specify these values using their respective setter methods as shown below —

```
//setting the level property
glow.setLevel(0.9);
```

Adding Effect to the Node

Finally, you can apply the required effect to the node using the **setEffect()** method. For example: To set the glow effect to the image node, you need to pass the object of the Glow class to this method as follows:

```
imageView.setEffect(glow);
```

JavaFX Effects: The following table gives you the list of various effects (classes) provided by JavaFX. These classes exist in the package called **javafx.scene.effect**.

S. No.	Shape and Description
1	**ColorAdjust:** You can adjust the color of an image by applying the color adjust effect to it. This includes the adjustment of the **hue**, **saturation**, **brightness** and **contrast** on each pixel. The class named **ColorAdjust** of the package **javafx.scene.effect** represents the color adjust effect.

2	**ColorInput:** Color input effect gives the same output as drawing a rectangle and filling it with color. Unlike other effects, if this effect is applied to any node, it displays only a rectangular box (not the node). This effect is mostly used to pass as an input for other effects. The class named **ColorInput** of the package **javafx.scene.effect** represents the color input effect.
3	**ImageInput:** Image input effect in JavaFX just embeds an image to the JavaFX screen. Just like Color Input effect (It is used to pass the specified colored rectangular region as input to other effect), Image Input effect is used to pass the specified image as an input to another effect. The class named **ImageInput** of the package **javafx.scene.effect** represents the Image Input effect.
4	**Blend:** In general, blend means mixture of two or more different things or substances. If we apply this blend effect, it takes the pixels of two different inputs, at the same location and produces combined output based on the **blend mode**. The class named **Blend** of the package **javafx.scene.effect** represents the blend effect.
5	**Bloom:** On applying bloom effect, pixels in some portions of the node are made to glow. The class named **Bloom** of the package **javafx.scene.effect** represents the bloom effect.
6	**Glow:** Just like bloom, the Glow effect makes the given input image to glow, this effect makes the bright pixels of the input brighter. The class named **Glow** of the package **javafx.scene.effect** represents the glow effect.
7	**BoxBlur:** On applying this blur effect to a node, it is made unclear. Box blur is a kind of blur effect provided by JavaFX. In this effect, when we apply blur to a node, a simple box filter is used. The class named **BoxBlur** of the package **javafx.scene.effect** represents the boxblur effect.

8	**GaussianBlur:** Just like Box Blur Gaussian is an effect to blur the nodes in JavaFX. The only difference in the **Gaussian Blur effect** is that a Gaussian convolution kernel is used to produce a blurring effect. The class named GaussianBlur of the package **javafx.scene.effect** represents the Gaussian Blur effect.
9	**MotionBlur:** Just like Gaussian Effects, Motion Blur is an effect to blur the nodes in JavaFX. It also uses a Gaussian convolution kernel to produce a blurring effect, but the difference is in this effect the Gaussian convolution kernel is used with a specified angle. The class named **MotionBlur** of the package **javafx.scene.effect** represents the Motion Blur effect.
10	**Reflection:** On applying the reflection effect to a node in JavaFX, a reflection of it is added at the bottom of the node. The class named **Reflection** of the package **javafx.scene.effect** represents the reflection effect.
11	**SepiaTone:** On applying the Sepia tone effect to a node in JavaFX (image in general), it is toned with a reddish brown color. The class named **SepiaTone** of the package **javafx.scene.effect** represents the sepia tone effect.
12	**Shadow:** This effect creates a duplicate of the specified node with blurry edges. The class named **Shadow** of the package **javafx.scene.effect** represents the sepia tone effect.
13	**DropShadow:** On applying this effect to a node, a shadow will be created behind the specified node. The class named **DropShadow** of the package **javafx.scene.effect** represents the drop shadow effect.
14	**InnerShadow:** On applying this effect to a node, a shadow will be created inside the edges of the node. The class named **InnerShadow** of the package **javafx.scene.effect** represents the inner shadow effect

15	**Lighting:** The lighting effect is used to simulate a light from a light source. There are different kinds of light sources namely **point**, **distant** and **spot**. The class named **Lighting** of the package **javafx.scene.effect** represents the lighting effect.
16	**Light.Distant:** On applying this effect to a node, a light is simulated on it, as if it is being generated by a distant light source. **Distant light source:** A source which is at a far distance from the node. In here, the light is attenuated in one direction from the source. The class named **Light.Distant** of the package **javafx.scene.effect** represents the distant light source.
17	**Light.Spot:** On applying this effect to a node, a light is simulated on it, as if it is being generated by a spot light. **Spot light source:** The light from this source attenuates in all directions. The intensity of the light depends on the distance of the object from the source. The class named **Light.Spot** of the package **javafx.scene.effect** represents the distant light source.
18	**Point.Spot:** On applying this effect to a node, a light is simulated on it, as if it is being generated by a point light source. **Point light source:** The light from this source attenuates in all directions from a single point. The intensity of the light depends on the distance of the object from the source. The class named **Point.Spot** of the package **javafx.scene.effect** represents the point light.

Color Adjust Effect

You can adjust the color of an image by applying the color adjust effect to it. This includes the adjustment of the **Hue**, **Saturation**, **Brightness** and **Contrast** on each pixel.

The class named **ColorAdjust** of the package **javafx.scene.effect** represents the color adjust effect, this class contains five properties namely –

- **input:** This property is of the Effect type and it represents an input to the color adjust effect.
- **brightness**: This property is of Double type and it represents the brightness adjustment value for this effect.
- **contrast**: This property is of Double type and it represents the contrast adjustment value for this effect.

- **hue**: This property is of Double type and it represents the hue adjustment value for this effect.
- **saturation:** This property is of Double type and it represents the saturation adjustment value for this effect.

Example

The following program is an example of demonstrating the color adjust effect. In here, we are embedding an image (Tutorialspoint Logo) in JavaFX scene using **Image** and **ImageView** classes. This is being done at the position 100, 70 and with a fit height and fit width of 200 and 400 respectively.

We are adjusting the color of this image using the color adjust effect. With **contrast**, **hue**, **brightness** and **saturation** values as 0.4. -0.05, 0.9, 0.8.

Save this code in a file with the name **ColorAdjustEffectExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.effect.ColorAdjust;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.stage.Stage;

public class ColorAdjustEffectExample extends Application {

   @Override
   public void start(Stage stage) {

      //Creating an image
      Image image = new

Image("http://www.tutorialspoint.com/green/images/logo.png");
```

```
//Setting the image view
ImageView imageView = new ImageView(image);
//Setting the position of the image
imageView.setX(100);

imageView.setY(70);
//setting the fit height and width of the image view
imageView.setFitHeight(200);
imageView.setFitWidth(400);
//Setting the preserve ratio of the image view
imageView.setPreserveRatio(true);

//Instantiating the ColorAdjust class
ColorAdjust colorAdjust = new ColorAdjust();
//Setting the contrast value
colorAdjust.setContrast(0.4);
//Setting the hue value
colorAdjust.setHue(-0.05);
//Setting the brightness value
colorAdjust.setBrightness(0.9);
//Setting the saturation value
colorAdjust.setSaturation(0.8);

//Applying coloradjust effect to the ImageView node
imageView.setEffect(colorAdjust);

//Creating a Group object
Group root = new Group(imageView);

//Creating a scene object
Scene scene = new Scene(root, 600, 300);

//Setting title to the Stage
stage.setTitle("Coloradjust effect example");
```

```
      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage

     stage.show();
   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac ColorAdjustEffectExample.java
java ColorAdjustEffectExample
```

On executing, the above program generates a JavaFX window as shown below.

Color Input Effect

Color Input Effect gives the same output as drawing a rectangle and filling it with color. Unlike other effects, if this effect is applied to any node, it displays only a rectangular box (not the node). This effect is mostly used to pass as an input for other effects.

For example, while applying the blend effect, it requires an object of effect type as input. There we can pass this as an input.

The class named **ColorInput** of the package **javafx.scene.effect** represents the color input effect. This class contains four properties namely –

- **x**: This property is of double type; it represents the x coordinate of the position of the color input.
- **y**: This property is of double type; it represents the y coordinate of the position of the color input.
- **height**: This property is of double type; it represents the height of the region that is to be filled with color.
- **width:** This property is of double type; it represents the width of the region that is to be filled with color.
- **paint:** This property is of Paint type; it represents the color with which the input region is to be filled.

Example

Following is an example demonstrating the color input effect. In here, we are creating a color input of the dimensions 50, 400 (height, width) at the position 50, 140, and filling it with the color CHOCOLATE.

We are creating rectangle and applying this effect to it. Save this code in a file with the name **ColorInputEffectExample.java**.

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

public class ColorInputEffectExample extends Application {

   @Override

   public void start(Stage stage) {

      //creating a rectangle
      Rectangle rectangle = new Rectangle();
```

```
      //Instantiating the Colorinput class
      ColorInput colorInput = new ColorInput();
      //Setting the coordinates of the color input
      colorInput.setX(50);
      colorInput.setY(140);
      //Setting the height of the region of the collor input
      colorInput.setHeight(50);
      //Setting the width of the region of the collor input
      colorInput.setWidth(400);
      //Setting the color the collor input
      colorInput.setPaint(Color.CHOCOLATE);

      //Applying coloradjust effect to the Rectangle
      rectangle.setEffect(colorInput);

      //Creating a Group object
      Group root = new Group(rectangle);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Sample Application");

      //Adding scene to the stage
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }

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

```
}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac ColorInputEffectExample.java
java ColorInputEffectExample
```

On executing, the above program generates a JavaFX window as shown below.

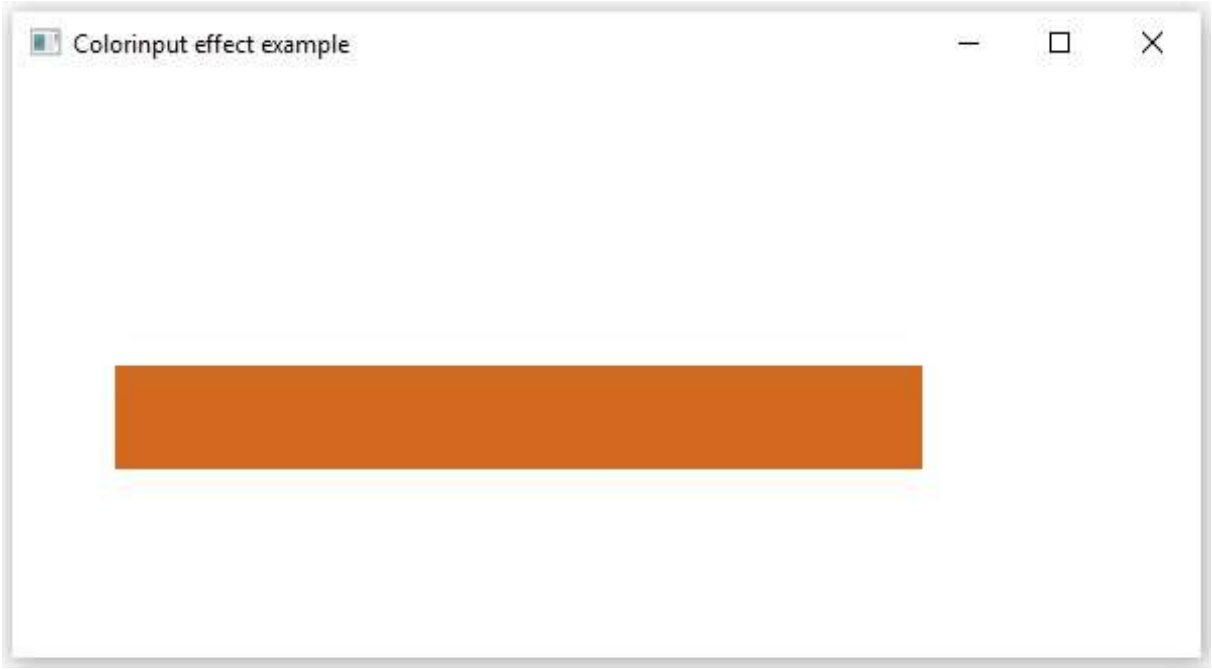


Image Input Effect

Image input effect in JavaFX just embeds an image to the JavaFX screen. Just like in the Color Input effect, it is used to pass the specified colored rectangular region as an input to another effect. An Image Input effect is used to pass the specified image as an input to another effect.

On applying this effect, the image specified will not be modified. This effect is applied to any node.

The class named **ImageInput** of the package **javafx.scene.effect** represents the Image Input effect, this class contains three properties, which are –

- **x**: This property is of Double type; it represents the x coordinate of the position of the source image.
- **y**: This property is of Double type; it represents the y coordinate of the position of the source image.
- **source**: This property is of Image type; it represents image that is to be used as a source to this effect. (Passed as input)

Example

The following program is an example demonstrating the Image input effect. In here, we are creating an image input at the position 150, 100, and taking the following image (tutorialspoint logo) as a source for this effect.

We are creating a rectangle and applying this effect to it. Save this code in a file with the name **ImageInputEffectExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.effect.ImageInput;
import javafx.scene.image.Image;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public class ImageInputEffectExample extends Application {

   @Override

   public void start(Stage stage) {

      //Creating an image
      Image image = new
Image("http://www.tutorialspoint.com/green/images/logo.png");

      //Instantiating the Rectangle class
      Rectangle rectangle = new Rectangle();



      //Instantiating the ImageInput class
      ImageInput imageInput = new ImageInput();
      //Setting the position of the image
```

```
        imageInput.setX(150);
        imageInput.setY(100);
        //Setting source for image input
        imageInput.setSource(image);

        //Applying image input effect to the rectangle node
        rectangle.setEffect(imageInput);

        //Creating a Group object
        Group root = new Group(rectangle);

        //Creating a scene object
        Scene scene = new Scene(root, 600, 300);

        //Setting title to the Stage
        stage.setTitle("Sample Application");

        //Adding scene to the stage
        stage.setScene(scene);

        //Displaying the contents of the stage
        stage.show();
    }

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

Compile and execute the saved java file from the command prompt using the following commands.

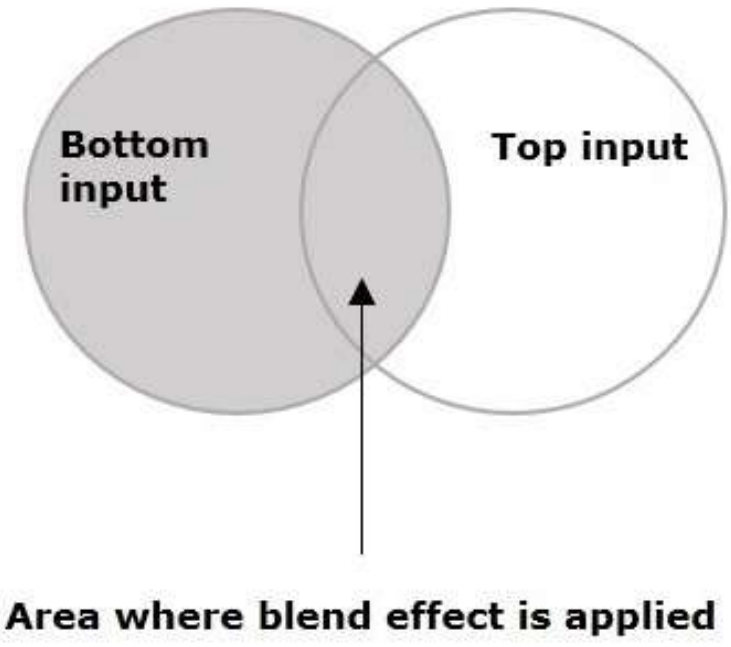
```
javac ImageInputEffectExample.java
java ImageInputEffectExample
```

On executing, the above program generates a JavaFX window as shown below.

Blend Effect

In general, blend means mixture of two or more different things or substances. If we apply the blend effect, it will take the pixels of two different inputs. This will be done at the same location and it produces a combined output based on the **blend mode**.

For example, if we draw two objects the top object covers the bottom one. On applying the blend effect, the pixels of the two objects in the overlap area are combined and displayed based on the input mode.



The class named **Blend** of the package **javafx.scene.effect** represents the blend effect, this class contains four properties, which are –

- **bottomInput**: This property is of the type Effect and it represents the bottom input to the blend effect.
- **topInput**: This property is of the type Effect and it represents the top input to the blend effect.
- **opacity**: This property is of double type and it represents the opacity value modulated with the top input.
- **mode:** This property is of the type BlendMode and it represents the mode used to blend the two inputs together.

Example

Following is an example demonstrating the blend effect. In here, we are drawing a circle filled with BROWN color, on top of it lies a BLUEVIOLET ColorInput.

We have applied the blend effect choosing a multiply mode In the overlap area, the colors of the two objects were multiplied and displayed.

Save this code in a file with the name **BlendEffectExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.shape.Circle;
import javafx.scene.effect.Blend;
import javafx.scene.effect.BlendMode;
import javafx.scene.effect.ColorInput;
import javafx.scene.paint.Color;

public class BlendEffectExample extends Application {

   @Override
   public void start(Stage stage) {

      //Drawing a Circle
      Circle circle = new Circle();
      //Setting the center of the Circle
      circle.setCenterX(75.0f);
      circle.setCenterY(75.0f);
```

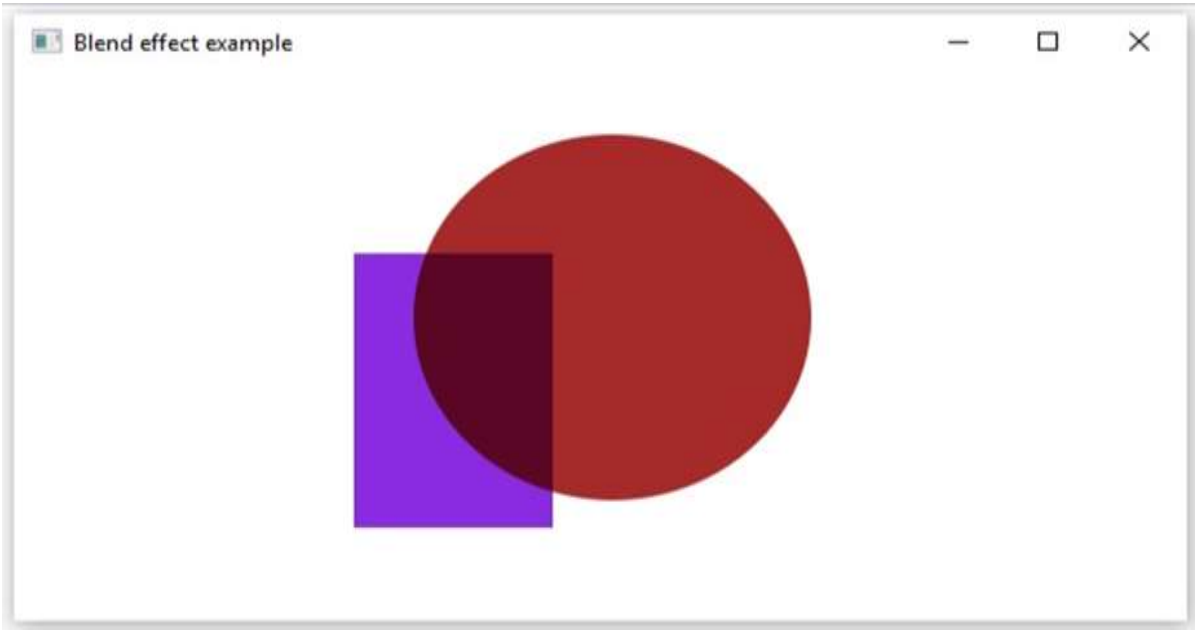
```
      //Settinmg radius of the circle
      circle.setRadius(30.0f);
      //Setting the fill color of the circle
      circle.setFill(Color.BROWN);

      //Instantiating the blend class
      Blend blend = new Blend();
      //Preparing the to input object
      ColorInput topInput = new ColorInput(35, 30, 75, 40, Color.BLUEVIOLET);
      //setting the top input to the blend object
      blend.setTopInput(topInput);
      //setting the belnd mode
      blend.setMode(BlendMode.SRC_OVER);

      //Applying the blend effect to circle

      circle.setEffect(blend);

      //Creating a Group object
      Group root = new Group(circle);

      //Creating a scene object
      Scene scene = new Scene(root, 150, 150);

      //Setting title to the Stage
      stage.setTitle("Blend Example");

      //Adding scene to the stage
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

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

```
    }
}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac BlendEffectExample.java
java BlendEffectExample
```

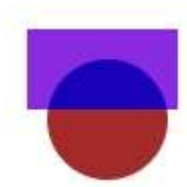
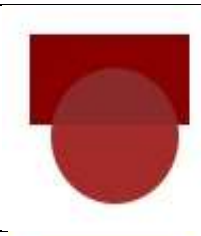
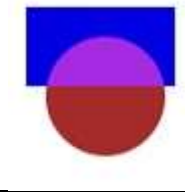
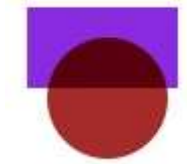
On executing, the above program generates a JavaFX window as shown below.

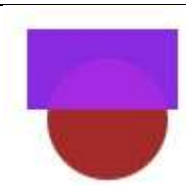
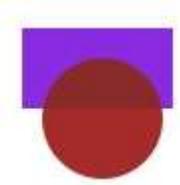
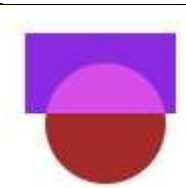
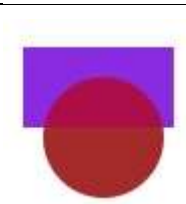
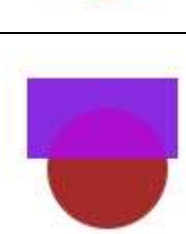
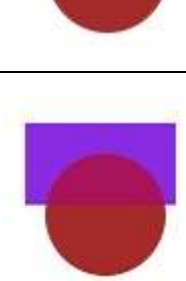
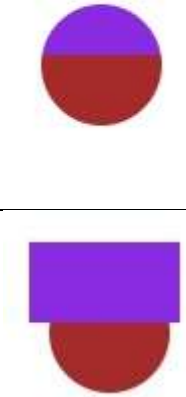


Blend Modes

In addition to the multiply mode, there are various blend modes in the Blend class. The following table lists all the various blend modes in JavaFX.

S. No.	Modes and description	Output
1	**ADD:** In this mode, the color values of the top and bottom inputs are added and displayed.	
2	**MULTIPLY:** In this mode, the color values of the top and bottom inputs are multiplied and displayed.	

3	**DIFFERENCE:** In this mode, among the color values of the top and bottom inputs, the darker one is subtracted from the lighter one and displayed.	
4	**RED:** In this mode, the red components of the bottom input were replaced by the red components of the top input.	
5	**BLUE:** In this mode, the blue components of the bottom input were replaced by the blue components of the top input.	
6	**GREEN:** In this mode, the green components of the bottom input were replaced by the green components of the top input.	
7	**EXCLUSION:** In this mode, the color components of the two inputs were multiplied and doubled. Then they are subtracted from the sum of the color components of the bottom input. The resultant is then displayed.	
8	**COLOR_BURN:** In this mode, the inverse of the bottom input color component was divided by the top input color component. Thus, the obtained value is inverted and displayed.	
9	**COLOR_DODGE:** In this mode, the bottom input color components were divided by the inverse of the top input color components and thus obtained value is inverted and displayed.	

10	**LIGHTEN:** In this mode, the lighter color component, among the both inputs are displayed.	
11	**DARKEN:** In this mode, the darker color component, among the top and bottom inputs is displayed.	
12	**SCREEN:** In this mode, the color components of the top and bottom inputs were inverted, multiplied and thus obtained value is inverted and displayed.	
13	**OVERLAY:** In this mode, based on the bottom input color, the color components of the two input values were multiplied or screened and the resultant is displayed.	
14	**HARD_LIGHT:** In this mode, based on the top input color, the color components of the two input values were multiplied or screened and the resultant is displayed.	
15	**SOFT_LIGHT:** In this mode, based on the top input color, the color components of the two input values were softened or lightened and the resultant is displayed.	
16	**SRC_ATOP:** In this mode, the over lapping area is filled with the color component of the bottom input. While the non-overlapping area is filled with the color component of the top input.	
17	**SRC_OVER:** In this mode, the top input is drawn over the bottom input	

Bloom Effect

On applying the bloom effect, pixels in some portions of the node are made to glow.

The class named **Bloom** of the package **javafx.scene.effect** represents the bloom effect. This class contains two properties, which are –

- **input**: This property is of the type Effect and it represents an input to the bloom effect.
- **threshold**: This property is of the type double; this represents a threshold value of luminosity of the pixels of the node. All those pixels having luminosity greater than equal to this value are made to glow. The range of the threshold value is 0.0 to 1.0.

Example

Following is an example demonstrating the bloom effect. We will be drawing a text "Welcome to Tutorialspoint" and applying the bloom effect to it with a threshold value 1.0.

Save this code in a file with the name **BloomEffectExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.effect.Bloom;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;

public class BloomEffectExample extends Application {

   @Override
   public void start(Stage stage) {

      //Creating a Text object
      Text text = new Text();
      //Setting font to the text
```

```
text.setFont(Font.font(null, FontWeight.BOLD, 40));
//setting the position of the text

text.setX(60);
text.setY(150);
//Setting the text to be embedded.
text.setText("Welcome to Tutorialspoint");
//Setting the color of the text
text.setFill(Color.DARKSEAGREEN);
```

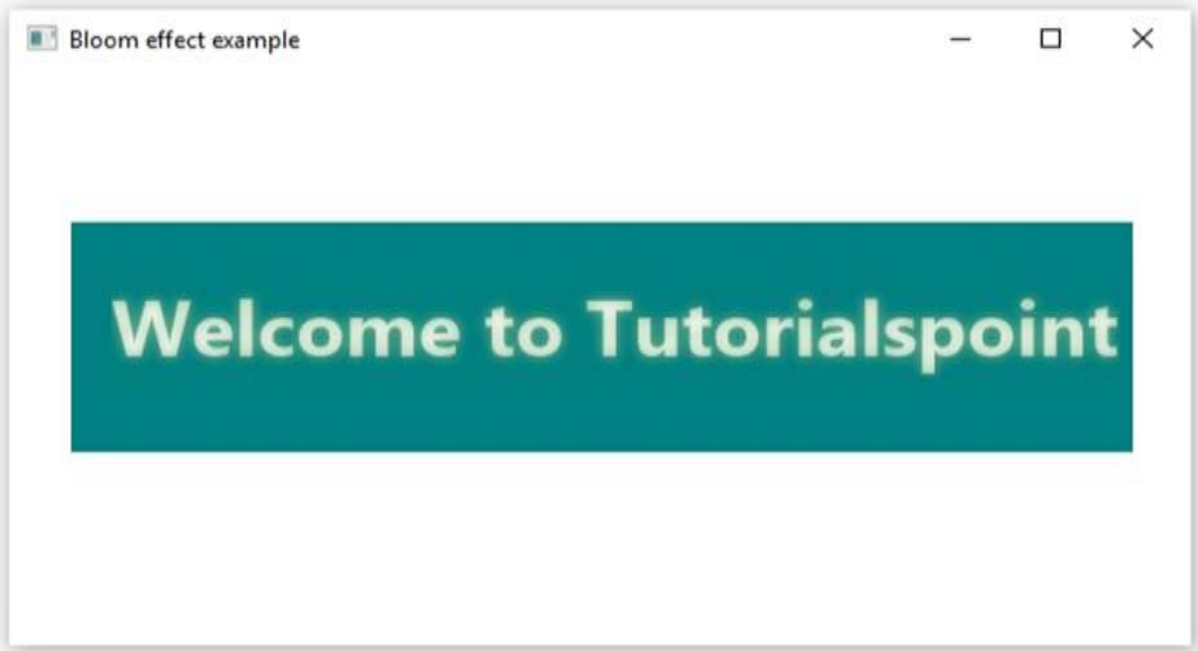
```
//Instantiating the Rectangle class
Rectangle rectangle = new Rectangle();
//Setting the position of the rectangle
rectangle.setX(50.0f);
rectangle.setY(80.0f);
//Setting the width of the rectangle
rectangle.setWidth(500.0f);
//Setting the height of the rectangle
rectangle.setHeight(120.0f);
//Setting the color of the rectangle
rectangle.setFill(Color.TEAL);

//Instantiating the Bloom class
Bloom bloom = new Bloom();
//setting threshold for bloom
bloom.setThreshold(0.1);

//Applying bloom effect to text
text.setEffect(bloom);

//Creating a Group object
Group root = new Group(rectangle, text);

//Creating a scene object
Scene scene = new Scene(root, 600, 300);
```

```
      //Setting title to the Stage
      stage.setTitle("Sample Application");

      //Adding scene to the stage
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }
   public static void main(String args[]){
      launch(args);
   }
}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac BloomEffectExample.java
java BloomEffectExample
```

On executing, the above program generates a JavaFX window as shown below.



Glow Effect

Just like the Bloom Effect, the Glow Effect also makes the given input image to glow. This effect makes the pixels of the input much brighter.

The class named **Glow** of the package **javafx.scene.effect** represents the glow effect. This class contains two properties namely –

- **input**: This property is of the type Effect and it represents an input to the glow effect.
- **level**: This property is of the type double; it represents intensity of the glow. The range of the level value is 0.0 to 1.0.

Example

The following program is an example demonstrating the Glow Effect of JavaFX. In here, we are embedding the following image (Tutorialspoint Logo) in JavaFX scene using **Image** and **ImageView** classes. This will be done at the position 100, 70 and with fit height and fit width 200 and 400 respectively.

To this image, we are applying the Glow Effect with the level value 0.9. Save this code in a file with the name **GlowEffectExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.effect.Glow;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.stage.Stage;

public class GlowEffectExample extends Application {

   @Override
   public void start(Stage stage) {
      //Creating an image
      Image image = new
Image("http://www.tutorialspoint.com/green/images/logo.png");
```

```
        //Setting the image view
        ImageView imageView = new ImageView(image);
        //setting the fith width of the image view
        imageView.setFitWidth(200);

        //Setting the preserve ratio of the image view
        imageView.setPreserveRatio(true);

        //Instantiating the Glow class
        Glow glow = new Glow();
        //setting level of the glow effect
        glow.setLevel(0.9);



        //Applying bloom effect to text
        imageView.setEffect(glow);

        //Creating a Group object
        Group root = new Group(imageView);

        //Creating a scene object
        Scene scene = new Scene(root, 600, 300);

        //Setting title to the Stage
        stage.setTitle("Sample Application");

        //Adding scene to the stage
        stage.setScene(scene);

        //Displaying the contents of the stage
        stage.show();
    }

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

```
    }
}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac GlowEffectExample.java
java GlowEffectExample
```

On executing, the above program generates a JavaFX window as shown below.

Box Blur Effect

In general, Blur means becoming unclear, on applying blur effect to a node it is made unclear. Box Blur is a kind of blur effect provided by JavaFX. In this effect, to apply blur to node, a simple box filter is used.

The class named **BoxBlur** of the package **javafx.scene.effect** represents the BoxBlur effect, this class contains four properties, which are –

- **height**: This property is of double type representing the vertical size of the effect.
- **width**: This property is of double type representing the horizontal size of the effect.

- **input**: This property is of the type effect and it represents an input to the BoxBlur effect.

- **iterations**: This property is of an integer type representing the number of iterations of the effect, which are to be applied on the node. This is done to improve its quality or smoothness.

Example

Following is an example demonstrating the box blur effect. In here, we are drawing the text "Welcome to Tutorialspoint" filled with DARKSEAGREEN color and applying the Box Blur effect to it.

Save this code in a file with the name **BoxBlurEffectExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.effect.BoxBlur;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;

public class BoxBlurEffectExample extends Application {

   @Override
   public void start(Stage stage) {

      //Creating a Text object
      Text text = new Text();
      //Setting font to the text
      text.setFont(Font.font(null, FontWeight.BOLD, 40));
      //setting the position of the text
      text.setX(60);
      text.setY(150);
      //Setting the text to be added.
      text.setText("Welcome to Tutorialspoint");
      //Setting the color of the text
      text.setFill(Color.DARKSEAGREEN);
```

```
      //Instantiating the BoxBlur class
      BoxBlur boxblur = new BoxBlur();
      //Setting the width of the box filter
      boxblur.setWidth(8.0f);

      //Setting the height of the box filter
      boxblur.setHeight(3.0f);
      //Setting the no of iterations
      boxblur.setIterations(3);

      //Applying BoxBlur effect to the text
      text.setEffect(boxblur);

      //Creating a Group object
      Group root = new Group(text);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Sample Application");

      //Adding scene to the stage
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac BoxBlurEffectExample.java
java BoxBlurEffectExample
```

On executing, the above program generates a JavaFX window as shown below.

Gaussian Blur Effect

Just like Box Blur, Gaussian is an effect to blur the nodes in JavaFX. The only difference is that in **Gaussian Blur Effect**, a Gaussian convolution kernel is used to produce the blurring effect.

The class named GaussianBlur of the package **javafx.scene.effect** represents the Gaussian Blur Effect, this class contains two properties, which are –

- **input**: This property is of the type Effect and it represents an input to the box blur effect.

- **radius**: This property is of a double type representing the radius with which the **Gaussian Blur effect** is to be applied. The blur effect is directly proportional to radius.

Example

The following program is an example demonstrating the Gaussian blur effect. In this, we are drawing a text "Welcome to Tutorialspoint" filled with DARKSEAGREEN color and applying the Gaussian Blur Effect to it.

Save this code in a file with the name **GaussianBlurEffectExample.java.**

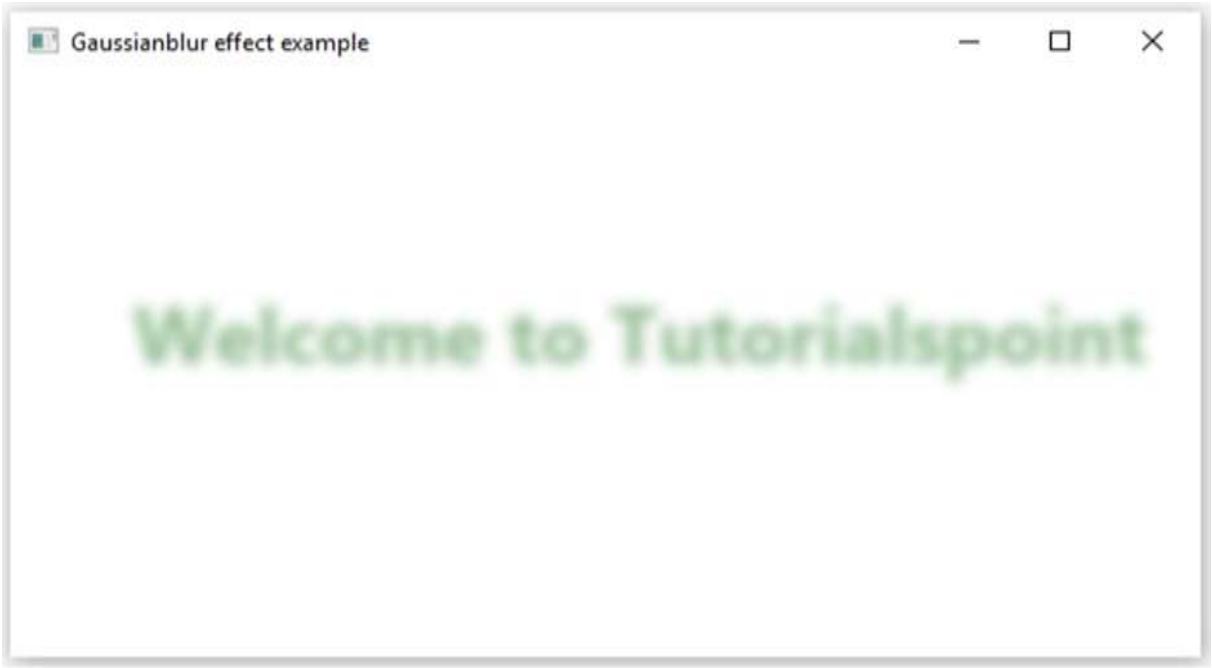
```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.scene.effect.GaussianBlur;

public class GaussianBlurEffectExample extends Application {

   @Override
   public void start(Stage stage) {

      //Creating a Text object
      Text text = new Text();
      //Setting font to the text
      text.setFont(Font.font(null, FontWeight.BOLD, 40));
      //setting the position of the text
      text.setX(60);
      text.setY(150);
      //Setting the text to be added.
      text.setText("Welcome to Tutorialspoint");
      //Setting the color of the text
      text.setFill(Color.DARKSEAGREEN);

      //Instantiating the GaussianBlur class
      GaussianBlur gaussianBlur = new GaussianBlur();
```

```
      //Setting the radious to apply the Gaussian Blur effect
      gaussianBlur.setRadius(10.5);

      //Applying Gaussian Blur effect to the text
      text.setEffect(gaussianBlur);

      //Creating a Group object
      Group root = new Group(text);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);
      //Setting title to the Stage
      stage.setTitle("Sample Application");

      //Adding scene to the stage
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }

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

   }

}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac GaussianBlurEffectExample.java
java GaussianBlurEffectExample
```

On executing, the above program generates a JavaFX window as shown below.



Motion Blur Effect

Just like Gaussian Effect, Motion Blur is an effect to blur the nodes in JavaFX. It also uses a Gaussian Convolution Kernel that helps in producing the blurring effect. The only difference between Gaussian Effect and Motion Blur is that the Gaussian Convolution Kernel is used with a specified angle.

As indicated by the name, on applying this effect by specifying some angle, the given input seems to you as if you are seeing it while it is in motion.

The class named **MotionBlur** of the package **javafx.scene.effect** represents the Motion Blur effect. This class contains three properties, which include –

- **input**: This property is of the type Effect and it represents an input to the box blur effect.
- **radius**: This property is of double type representing the radius with which the **Motion Blur Effect** is to be applied.
- **Angle:** This is a property of double type and it represents the angle of the motion effect in degrees.

Example

The following program is an example demonstrating the Motion Blur Effect. In here, we are drawing the text "Welcome to Tutorialspoint" filled with DARKSEAGREEN color and applying Motion Blur Effect to it with an angle of 45 degrees.

Save this code in a file with the name **MotionBlurEffectExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.scene.effect.MotionBlur;

public class MotionBlurEffectExample extends Application {

   @Override
   public void start(Stage stage) {

      //Creating a Text object
      Text text = new Text();
      //Setting font to the text
      text.setFont(Font.font(null, FontWeight.BOLD, 40));
      //setting the position of the text
      text.setX(60);
      text.setY(150);
      //Setting the text to be added.
      text.setText("Welcome to Tutorialspoint");
      //Setting the color of the text
      text.setFill(Color.DARKSEAGREEN);

      //Instantiating the MotionBlur class
      MotionBlur motionBlur = new MotionBlur();
      //Setting the radius to the effect
      motionBlur.setRadius(10.5);
      //Setting angle to the effect
      motionBlur.setAngle(45);

      //Applying MotionBlur effect to text
```

```
      text.setEffect(motionBlur);

      //Creating a Group object
      Group root = new Group(text);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Sample Application");

      //Adding scene to the stage
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac MotionBlurEffectExample.java
java MotionBlurEffectExample
```

On executing, the above program generates a JavaFX window as shown below.

Reflection Effect

On applying the reflection effect to a node in JavaFX, a reflection of it is added at the bottom of the node.

The class named **Reflection** of the package **javafx.scene.effect** represents the reflection effect. This class contains four properties, which are –

- **topOpacity**: This property is of double type representing the top extreme's opacity value of the reflection.
- **bottomOpacity**: This property is of double type representing the bottom extreme's opacity value of the reflection.
- **input**: This property is of the type Effect and it represents an input to the reflection effect.
- **topOffset**: This property is of double type representing the distance between the bottom of the input and the top of the reflection.
- **fraction:** This property is of double type representing the fraction of input that is visible in the output. The range of the fraction value is 0.0 to 1.0.

Example

The following program is an example demonstrating the reflection effect. In here, we are drawing the text "Welcome to Tutorialspoint" filled with DARKSEAGREEN color and applying the reflection effect to it.

Save this code in a file with the name **reflectionEffectExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.effect.Reflection;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;

public class ReflectionEffectExample extends Application {

   @Override
   public void start(Stage stage) {

      //Creating a Text object
      Text text = new Text();
      //Setting font to the text
      text.setFont(Font.font(null, FontWeight.BOLD, 40));
      //setting the position of the text
      text.setX(60);
      text.setY(150);
      //Setting the text to be embedded.
      text.setText("Welcome to Tutorialspoint");
      //Setting the color of the text
      text.setFill(Color.DARKSEAGREEN);

      //Instanting the reflection class
      Reflection reflection = new Reflection();
      //setting the bottom opcity of the reflection
      reflection.setBottomOpacity(0.0);
      //setting the top opcity of the reflection
      reflection.setTopOpacity(0.5);
      //setting the top offset of the reflection
      reflection.setTopOffset(0.0);
```

```
      //Setting the fraction of the reflection
      reflection.setFraction(0.7);

      //Applying reflection effect to the text
      text.setEffect(reflection);

      //Creating a Group object
      Group root = new Group(text);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Reflection effect example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac ReflectionEffectExample.java
java ReflectionEffectExample
```

On executing, the above program generates a javaFx window as shown below.

Sepia Tone Effect

On applying the Sepia Tone Effect to a node in JavaFX (image in general), it is toned with a reddish brown color.

The class named **SepiaTone** of the package **javafx.scene.effect** represents the sepia tone effect, this class contains two properties, which are –

- **level**: This property is of double type representing the intensity of this effect. The range of this property is 0.0 to 1.0.
- **input**: This property is of the type effect and it represents an input to the sepia tone effect.

Example

The following program is an example demonstrating the Sepia Tone Effect of JavaFX. In here, we are embedding the following image (tutorialspoint logo) in JavaFX scene using **Image** and **ImageView** classes. This is done at the position 100, 70 along with fit height and fit width 200 and 400 respectively.

To this image, we are applying the Sepia Tone Effect with the level value 0.9. Save this code in a file with name **SepiaToneEffectExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.effect.SepiaTone;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.stage.Stage;

public class SepiaToneEffectExample extends Application {
   @Override
   public void start(Stage stage) {

      //Creating an image
      Image image = new Image("http://www.tutorialspoint.com/images/tp-logo.gif");

      //Setting the image view
      ImageView imageView = new ImageView(image);
      //Setting the position of the image

      imageView.setX(150);
      imageView.setY(0);
```

```
      //setting the fit height and width of the image view
      imageView.setFitHeight(300);
      imageView.setFitWidth(400);
      //Setting the preserve ratio of the image view
      imageView.setPreserveRatio(true);

      //Instanting the SepiaTone class
      SepiaTone sepiaTone = new SepiaTone();
      //Setting the level of the effect
      sepiaTone.setLevel(0.8);

      //Applying SepiaTone effect to the image
      imageView.setEffect(sepiaTone);

      //Creating a Group object
      Group root = new Group(imageView);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Reflection effect example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac SepiaToneEffectExample.java
java SepiaToneEffectExample
```

On executing, the above program generates a JavaFX window as shown below.

Shadow Effect

This effect creates a duplicate of the specified node with blurry edges.

The class named **Shadow** of the package **javafx.scene.effect** represents the sepia tone effect. This class contains six properties, which are –

- **color**: This property is of Color type represents the color of the shadow.
- **blur type**: This property is of the BlurType and it represents the type of the blur effect used to blur the shadow.
- **radius:** This property is of the type double and it represents the radius of the shadow blur kernel.
- **width:** This property is of the type double and it represents the width of the shadow blur kernel.
- **height:** This property is of the type double and it represents the height of the shadow blur kernel.
- **input:** This property is of the type Effect and it represents an input to the shadow effect.

Example

The following program is an example demonstrating the shadow effect of JavaFX. In here, we are drawing the text “Welcome to Tutorialspoint”, and a circle in a scene.

We are applying the shadow effect with the Blur Type Gaussian with the Color Rosy Brown and Height, Width, Radius as 5.

Save this code in a file with the name **ShadowEffectExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.effect.BlurType;
import javafx.scene.effect.Shadow;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;

public class ShadowEffectExample extends Application {

   @Override
   public void start(Stage stage) {

      //Creating a Text object
      Text text = new Text();
      //Setting font to the text
      text.setFont(Font.font(null, FontWeight.BOLD, 40));

      //setting the position of the text
      text.setX(60);

      text.setY(50);
      //Setting the text to be embedded.
      text.setText("Welcome to Tutorialspoint");
      //Setting the color of the text
      text.setFill(Color.DARKSEAGREEN);
```

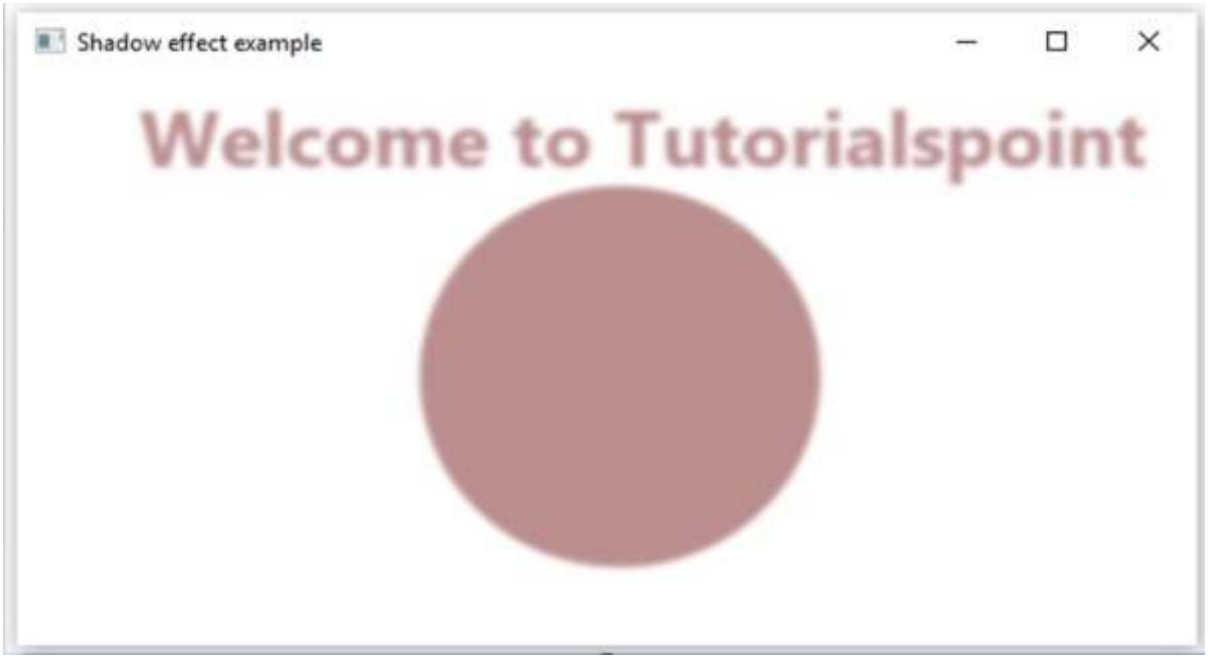
```
//Drawing a Circle
Circle circle = new Circle();
//Setting the center of the circle
circle.setCenterX(300.0f);
circle.setCenterY(160.0f);
//Setting the radius of the circle
circle.setRadius(100.0f);

//Instantiating the Shadow class
Shadow shadow = new Shadow();
//setting the type of blur for the shadow
shadow.setBlurType(BlurType.GAUSSIAN);
//Setting color of the shadow
shadow.setColor(Color.ROSYBROWN);
//Setting the height of the shadow
shadow.setHeight(5);
//Setting the width of the shadow
shadow.setWidth(5);
//Setting the radius of the shadow
shadow.setRadius(5);

//Applying shadow effect to the text
text.setEffect(shadow);
//Applying shadow effect to the circle
circle.setEffect(shadow);

//Creating a Group object


Group root = new Group(circle, text);

//Creating a scene object
Scene scene = new Scene(root, 600, 300);

//Setting title to the Stage
stage.setTitle("Bloom effect example");
```

```
      //Adding scene to the stage
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac ShadowEffectExample.java
java ShadowEffectExample
```

On executing, the above program generates a JavaFX window as shown below.



Drop Shadow Effect

On applying this effect to a node, a shadow will be created behind the specified node.

The class named **DropShadow** of the package **javafx.scene.effect** represents the drop shadow effect. This class contains nine properties, which are –

- **color**: This property is of Color type representing the color of the shadow.
- **blur Type**: This property is of the type – BlurType and it represents the type of the blur effect used to blur the shadow.
- **radius:** This property is of the type double and it represents the radius of the shadow blur kernel.
- **width:** This property is of the type double and it represents the width of the shadow blur kernel.
- **height:** This property is of the type double and it represents the height of the shadow blur kernel.
- **input:** This property is of the type Effect and it represents an input to the shadow effect.
- **spread:** This property is of the type double; it represents the spread of the shadow.
- **offsetX:** This property is of the type double and it represents the shadow offset in the x direction in pixels.
- **offset:** This property is of the type double and it represents the shadow offset in the y direction in pixels.

Example

The following program is an example demonstrating the drop shadow effect of JavaFX. In here, we are drawing a text "Welcome to Tutorialspoint" and a circle in a scene.

To these, we are applying the drop shadow effect. Save this code in a file with the name **DropShadowEffectExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.effect.BlurType;
import javafx.scene.effect.DropShadow;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;

public class DropShadowEffectExample extends Application {
```

```
@Override
public void start(Stage stage) {

   //Creating a Text object
   Text text = new Text();

   //Setting font to the text
   text.setFont(Font.font(null, FontWeight.BOLD, 40));

   //setting the position of the text
   text.setX(60);
   text.setY(50);

   //Setting the text to be embedded.

   text.setText("Welcome to Tutorialspoint");

   //Setting the color of the text
   text.setFill(Color.DARKSEAGREEN);

   //Drawing a Circle
   Circle circle = new Circle();
   //Setting the center of the circle
   circle.setCenterX(300.0f);
   circle.setCenterY(160.0f);
   //Setting the radius of the circle
   circle.setRadius(100.0f);

   //Instantiating the Shadow class
   DropShadow dropShadow = new DropShadow();
   //setting the type of blur for the shadow
   dropShadow.setBlurType(BlurType.GAUSSIAN);
   //Setting color of the shadow
   dropShadow.setColor(Color.ROSYBROWN);
   //Setting the height of the shadow
```

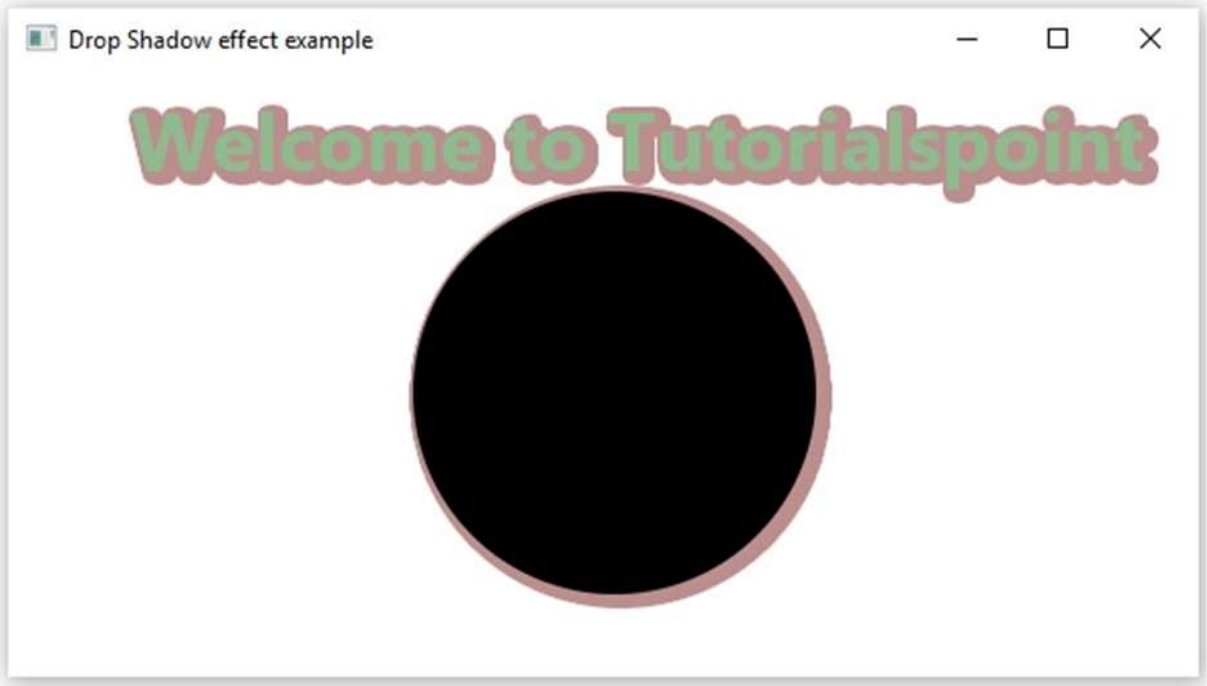
```
      dropShadow.setHeight(5);
      //Setting the width of the shadow
      dropShadow.setWidth(5);
      //Setting the radius of the shadow
      dropShadow.setRadius(5);
      //setting the foffset of the shadow
      dropShadow.setOffsetX(3);
      dropShadow.setOffsetY(2);
      //Setting the spread of the shadow
      dropShadow.setSpread(12);

      //Applying shadow effect to the text
      text.setEffect(dropShadow);
      //Applying shadow effect to the circle
      circle.setEffect(dropShadow);



      //Creating a Group object

      Group root = new Group(circle, text);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Drop Shadow effect example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }

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

```
    }
}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac DropShadowEffectExample.java
java DropShadowEffectExample
```

On executing, the above program generates a JavaFX window as shown below.



Inner Shadow Effect

On applying this effect to a node, a shadow will be created inside the edges of the node.

The class named **InnerShadow** of the package **javafx.scene.effect** represents the inner shadow effect. This class contains ten properties, which are –

- **color**: This property is of Color type representing the color of the shadow.
- **blur Type**: This property is of BlurType and it represents the type of blur effect used to blur the shadow.
- **radius:** This property is of the type double and it represents the radius of the shadow blur kernel.
- **width:** This property is of the type double and it represents the width of the shadow blur kernel.
- **height:** This property is of the type double and it represents the height of the shadow blur kernel.

- **input:** This property is of the type Effect and it represents an input to the shadow effect.
- **spread:** This property is of the type double; it represents the spread of the shadow.
- **offsetX:** This property is of the type double, it represents the shadow offset in the x direction, in pixels.
- **offsetY:** This property is of the type double, it represents the shadow offset in the y direction in pixels.
- **choke:** This property is of double type; it represents the choke of the shadow.

Example

The following program is an example demonstrating the inner shadow effect of JavaFX. In here, we are drawing a text "Welcome to Tutorialspoint", and a circle in a scene.

To these, we are applying the inner shadow effect. Save this code in a file with the name **InnerShadowEffectExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.effect.InnerShadow;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;

public class InnerShadowEffectExample extends Application {

   @Override
   public void start(Stage stage) {

      //Creating a Text object
      Text text = new Text();
      //Setting font to the text
      text.setFont(Font.font(null, FontWeight.BOLD, 40));
      //setting the position of the text
      text.setX(60);
```

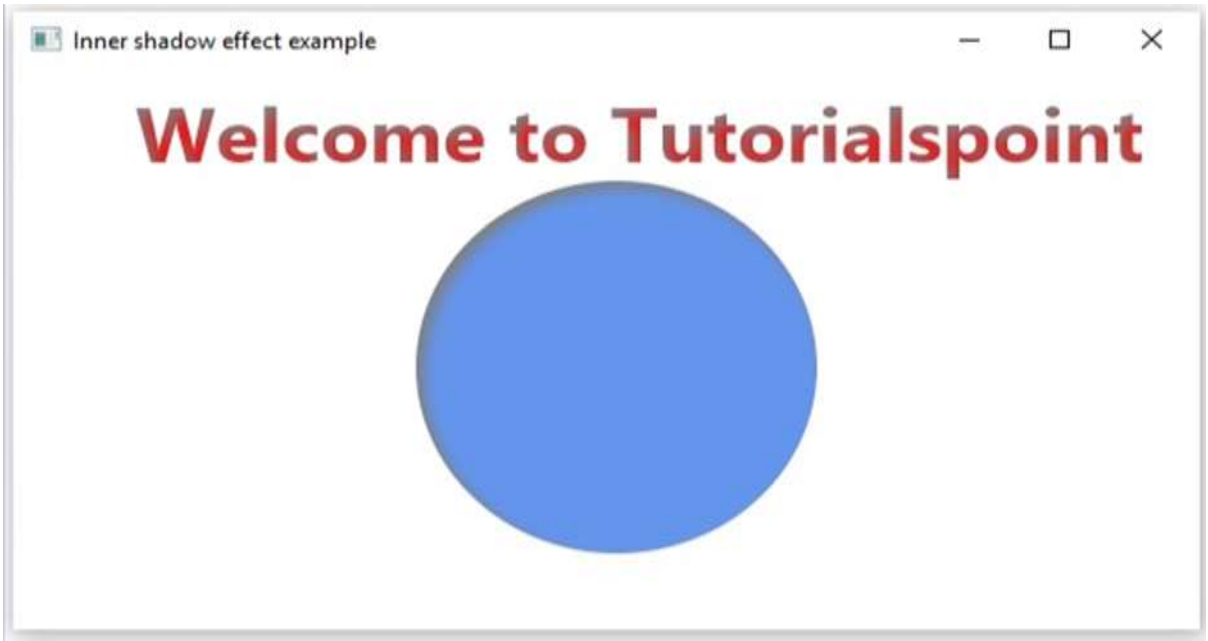
```
text.setY(50);
//Setting the text to be embedded.
text.setText("Welcome to Tutorialspoint");

//Setting the color of the text
text.setFill(Color.RED);

//Drawing a Circle
Circle circle = new Circle();
//Setting the center of the circle
circle.setCenterX(300.0f);
circle.setCenterY(160.0f);
//Setting the radius of the circle
circle.setRadius(100.0f);
//setting the fill color of the circle
circle.setFill(Color.CORNFLOWERBLUE);

//Instantiating the InnerShadow class
InnerShadow innerShadow = new InnerShadow();
//Setting the offset values of the inner shadow
innerShadow.setOffsetX(4);
innerShadow.setOffsetY(4);
//Setting the color of the inner shadow
innerShadow.setColor(Color.GRAY);

//Applying inner shadow effect to the text
text.setEffect(innerShadow);
//Applying inner shadow effect to the circle
circle.setEffect(innerShadow);

//Creating a Group object
Group root = new Group(text,circle);

//Creating a scene object
Scene scene = new Scene(root, 600, 300);
```

```
      //Setting title to the Stage
      stage.setTitle("Inner shadow effect example");


      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac InnerShadowEffectExample.java
java InnerShadowEffectExample
```

On executing, the above program generates a JavaFX window as shown below.



Lighting Effect (Default Source)

The lighting effect is used to simulate a light from a light source. There are different kinds of light sources which include – **Point**, **Distant** and **Spot**.

If we do not mention any source for lighting, it uses the default source of JavaFX.

The class named **Lighting** of the package **javafx.scene.effect** represents the lighting effect, this class contains ten properties, which are –

- **bumpInput**: This property is of the type Effect and it represents an optional bump map input to the lighting effect.
- **contentInput**: This property is of the type Effect and it represents a content input to the lighting effect.
- **diffuseConstant:** This property is of the type double and it represents the diffuse constant of the light.
- **SpecularConstant:** This property is of the type double and it represents the specular constant of the light.
- **SpecularExponent:** This property is of the type double and it represents the specular exponent of the light.
- **SurfaceScale:** This property is of the type double and it represents the surface scale factor of the light.

Example

The following program is an example demonstrating the lighting effect of JavaFX. In here we are drawing a text "Welcome to Tutorialspoint" and a circle in a scene.

To these, we are applying the lighting effect. In here, as we are not mentioning any source, JavaFX uses its default source.

Save this code in a file with the name **LightingEffectExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.effect.Lighting;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
import javafx.scene.text.Font;
```

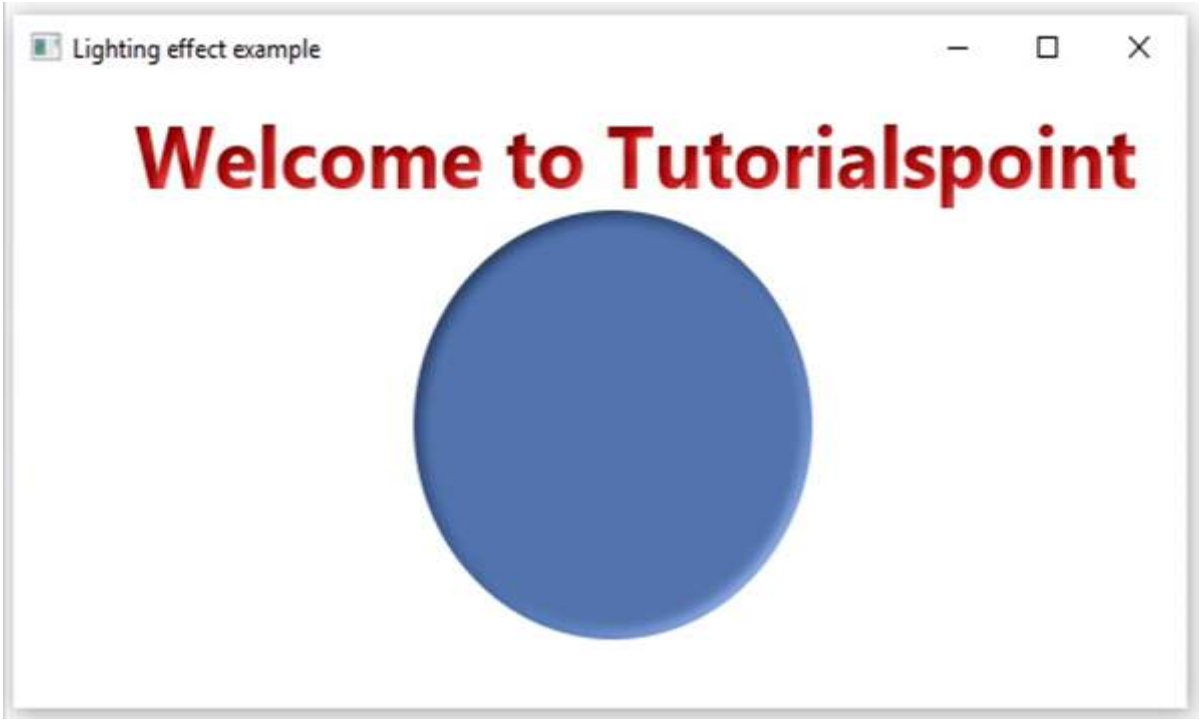
```
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;

public class LightingEffectExample extends Application {

   @Override
   public void start(Stage stage) {

      //Creating a Text object
      Text text = new Text();
      //Setting font to the text
      text.setFont(Font.font(null, FontWeight.BOLD, 40));
      //setting the position of the text
      text.setX(60);
      text.setY(50);
      //Setting the text to be embedded.
      text.setText("Welcome to Tutorialspoint");
      //Setting the color of the text
      text.setFill(Color.RED);

      //Drawing a Circle
      Circle circle = new Circle();
      //Setting the center of the circle
      circle.setCenterX(300.0f);
      circle.setCenterY(160.0f);
      //Setting the radius of the circle
      circle.setRadius(100.0f);
      //setting the fill color of the circle
      circle.setFill(Color.CORNFLOWERBLUE);

      //Instantiating the Lighting class
      Lighting lighting = new Lighting();

      //Applying lighting effect to the text
      text.setEffect(lighting);
      //Applying lighting effect to the circle
```

```
      circle.setEffect(lighting);

      //Creating a Group object
      Group root = new Group(text,circle);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Distant light effect example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }

   public static void main(String args[]){

      launch(args);
   }
}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac LightingEffectExample.java
java LightingEffectExample
```

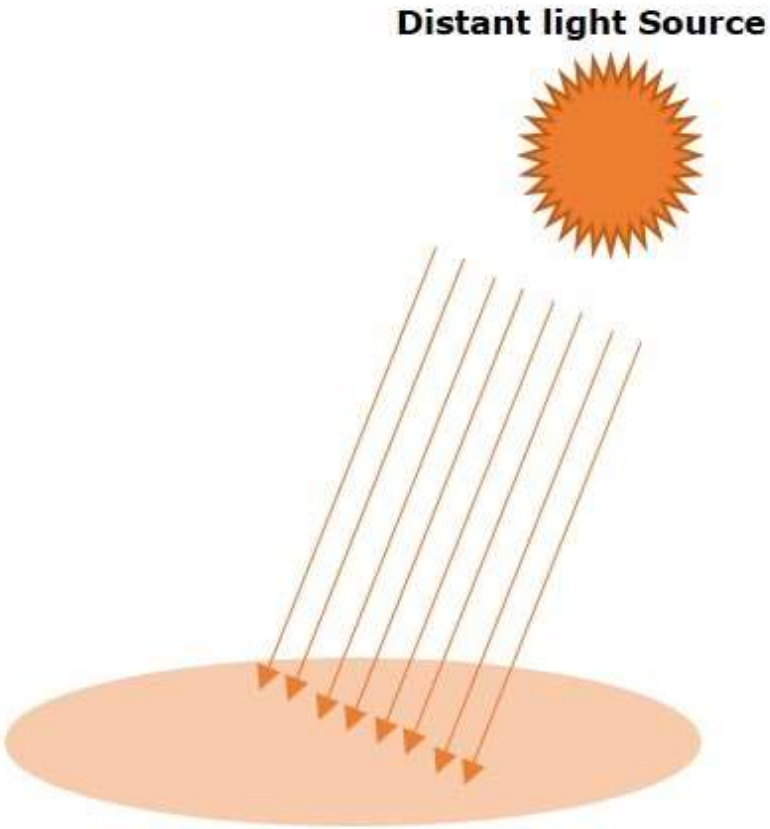
On executing, the above program generates a JavaFX window as shown below.



Lighting Effect (Distant Source)

On applying this effect to a node, a light is simulated on it, as if it is being generated by a distant light source.

Distant Light Source: A source which is at a far distance from the node. In here, the light is attenuated in one direction from the source.



The class named **Light.Distant** of the package **javafx.scene.effect** represents the distant light source. This class contains two properties, which include –

- **azimuth**: This property is of the type double and it represents the azimuth of the light.
- **elevation**: This property is of the type double and it represents the elevation of the light.

Example

The following program is an example demonstrating the lighting effect of JavaFX. In this, the source of light is a distant source. Here, we are drawing a text "Welcome to Tutorialspoint" and a circle in a scene.

To these, we are applying the lighting effect, where the light is being emitted by a distant source.

Save this code in a file with the name **DistantLightingExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.effect.Light;
import javafx.scene.effect.Lighting;
```

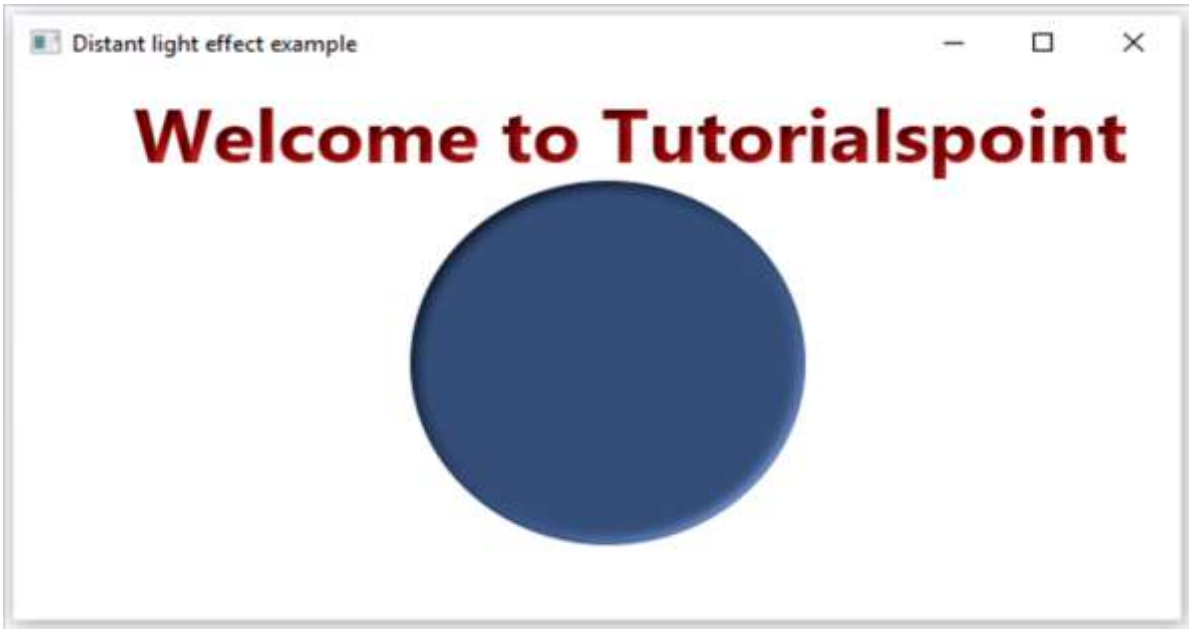
```
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;

public class DistantLightingExample extends Application {

   @Override
   public void start(Stage stage) {

      //Creating a Text object
      Text text = new Text();
      //Setting font to the text
      text.setFont(Font.font(null, FontWeight.BOLD, 40));
      //setting the position of the text
      text.setX(60);
      text.setY(50);
      //Setting the text to be embedded.
      text.setText("Welcome to Tutorialspoint");
      //Setting the color of the text
      text.setFill(Color.RED);

      //Drawing a Circle
      Circle circle = new Circle();
      //Setting the center of the circle
      circle.setCenterX(300.0f);
      circle.setCenterY(160.0f);
      //Setting the radius of the circle
      circle.setRadius(100.0f);
      //setting the fill color of the circle
      circle.setFill(Color.CORNFLOWERBLUE);

      //Instantiating the Light.Distant class
```

```
      Light.Distant light = new Light.Distant();
      //Setting the properties of the light source
      light.setAzimuth(45.0);
      light.setElevation(30.0);

      //Instantiating the Lighting class
      Lighting lighting = new Lighting();
      //Setting the source of the light
      lighting.setLight(light);

      //Applying the lighting effect to the text
      text.setEffect(lighting);
      //Applying the lighting effect to the circle
      circle.setEffect(lighting);

      //Creating a Group object
      Group root = new Group(text,circle);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Distant light effect example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }

   public static void main(String args[]){

      launch(args);
   }
}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac DistantLightingExample.java
java DistantLightingExample
```

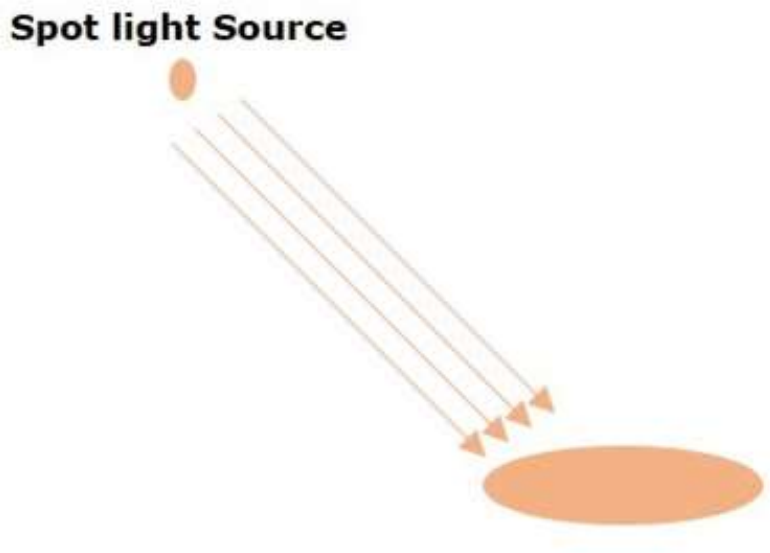
On executing, the above program generates a JavaFX window as shown below.



Lighting Effect (Spot Light as Source)

On applying this effect to a node, a light is simulated on it, as if it is being generated by a spot light.

Spot Light Source: The light from this source attenuates in all directions. The intensity of the light depends on the distance of the object from the source.



The class named **Light.Spot** of the package **javafx.scene.effect** represents the distant light source. This class contains four properties, which are –

- **pointsAtX**: This property is of the type double and it represents the x coordinate of the direction of vector for this light.
- **pointsAtY**: This property is of the type double and it represents the y coordinate of the direction of vector for this light.
- **pointsAtZ:** This property is of the type double and it represents the z coordinate of the direction of vector for this light.
- **specularExponent:** This property is of the type double and it represents the specular exponent, which controls the focus of this light source.

Example

The following program is an example demonstrating the lighting effect of JavaFX. In here, we are drawing a text "Welcome to Tutorialspoint" and a circle in a scene.

To these, we are applying the lighting effect, where the light is being emitted by a spotlight.

Save this code in a file with the name **SpotLightExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;

import javafx.scene.effect.Light;
```

```
import javafx.scene.effect.Lighting;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;

public class SpotLightExample extends Application {

   @Override
   public void start(Stage stage) {

      //Creating a Text object
      Text text = new Text();
      //Setting font to the text
      text.setFont(Font.font(null, FontWeight.BOLD, 40));
      //setting the position of the text
      text.setX(60);
      text.setY(50);
      //Setting the text to be embedded.
      text.setText("Welcome to Tutorialspoint");
      //Setting the color of the text
      text.setFill(Color.RED);

      //Drawing a Circle
      Circle circle = new Circle();
      //Setting the center of the circle
      circle.setCenterX(300.0f);
      circle.setCenterY(160.0f);
      //Setting the radius of the circle
      circle.setRadius(100.0f);
      //setting the fill color of the circle

      circle.setFill(Color.CORNFLOWERBLUE);
```

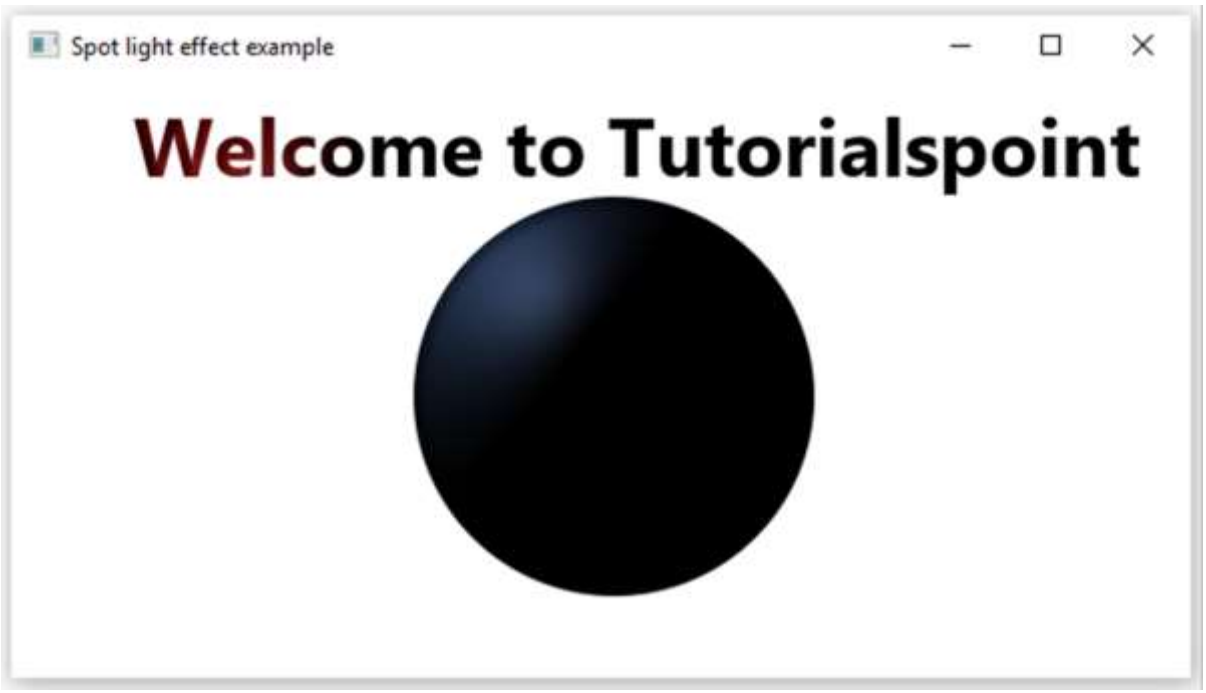
```
        //Instantiating the Light.Spot class
        Light.Spot light = new Light.Spot();
        //Setting the color of the light
        light.setColor(Color.GRAY);
        //setting the position of the light
        light.setX(70);
        light.setY(55);
        light.setZ(45);

        //Instantiating the Lighting class
        Lighting lighting = new Lighting();
        //Setting the light source
        lighting.setLight(light);

        //Applying lighting effect to the text
        text.setEffect(lighting);
        //Applying lighting effect to the circle
        circle.setEffect(lighting);

        //Creating a Group object
        Group root = new Group(text,circle);

        //Creating a scene object
        Scene scene = new Scene(root, 600, 300);

        //Setting title to the Stage
        stage.setTitle("Spot light effect example");

        //Adding scene to the satge
        stage.setScene(scene);

        //Displaying the contents of the stage
        stage.show();

    }
```

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

Compile and execute the saved java file from the command prompt using the following commands.

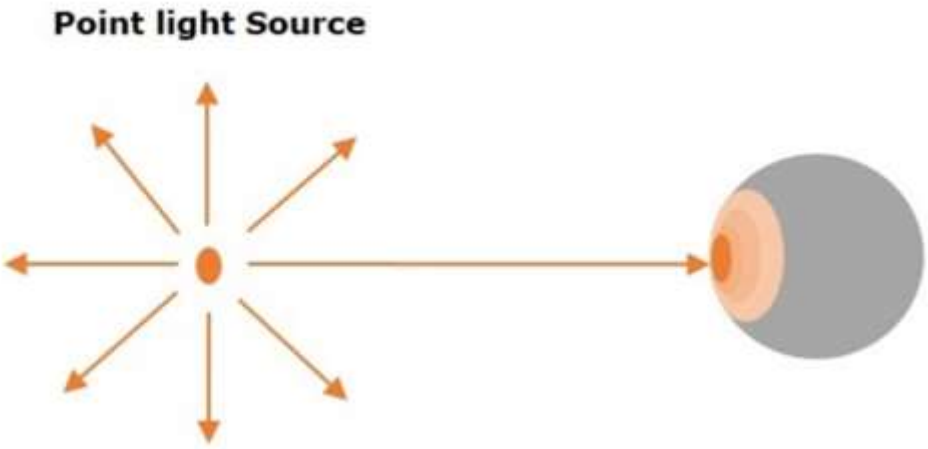
```
javac SpotLightExample.java
java SpotLightExample
```

On executing, the above program generates a JavaFX window as shown below.



Lighting Effect (Point Light as Source)

On applying this effect to a node, a light is simulated on it, as if it is being generated by a point light source.



Point Light Source: The light from this source attenuates in all directions from a single point the intensity of the light depends on the distance of the object from the source.

The class named **Point.Spot** of the package **javafx.scene.effect** represents the point light source. This class contains four properties, which include –

- **x:** This property is of the type double and it represents the x coordinate of the position of the light.
- **y:** This property is of the type double and it represents the y coordinate of the position of the light.
- **z:** This property is of the type double and it represents the z coordinate of the position of the light.

Example

The following program is an example demonstrating the lighting effect of JavaFX. In here, we are drawing a text "Welcome to Tutorialspoint" and a circle in a scene.

To these, we are applying the lighting effect, where the light is being emitted by a Point light source.

Save this code in a file with the name **PointLightExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.effect.Light;
import javafx.scene.effect.Lighting;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
import javafx.scene.text.Font;
```

```
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;

public class PointLightExample extends Application {

   @Override
   public void start(Stage stage) {

      //Creating a Text object
      Text text = new Text();
      //Setting font to the text
      text.setFont(Font.font(null, FontWeight.BOLD, 40));
      //setting the position of the text
      text.setX(60);
      text.setY(50);
      //Setting the text to be embedded.
      text.setText("Welcome to Tutorialspoint");
      //Setting the color of the text
      text.setFill(Color.RED);

      //Drawing a Circle
      Circle circle = new Circle();
      //Setting the center of the circle
      circle.setCenterX(300.0f);
      circle.setCenterY(160.0f);
      //Setting the radius of the circle
      circle.setRadius(100.0f);
      //setting the fill color of the circle
      circle.setFill(Color.CORNFLOWERBLUE);

      //instantiating the Light.Point class
      Light.Point light = new Light.Point();
      //Setting the color of the light

      light.setColor(Color.GRAY);
```

```
      //Setting the position of the light
      light.setX(70);
      light.setY(55);
      light.setZ(45);

      //Instantiating the Lighting class
      Lighting lighting = new Lighting();
      //Setting the light
      lighting.setLight(light);

      //Applying the Lighting effect to the text
      text.setEffect(lighting);
      //Applying the Lighting effect to the circle
      circle.setEffect(lighting);

      //Creating a Group object
      Group root = new Group(text,circle);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Point light effect example");

      //Adding scene to the stage
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac PointLightExample.java
java PointLightExample
```

On executing, the above program generates a JavaFX window as shown below.

8. JavaFX – Transformations

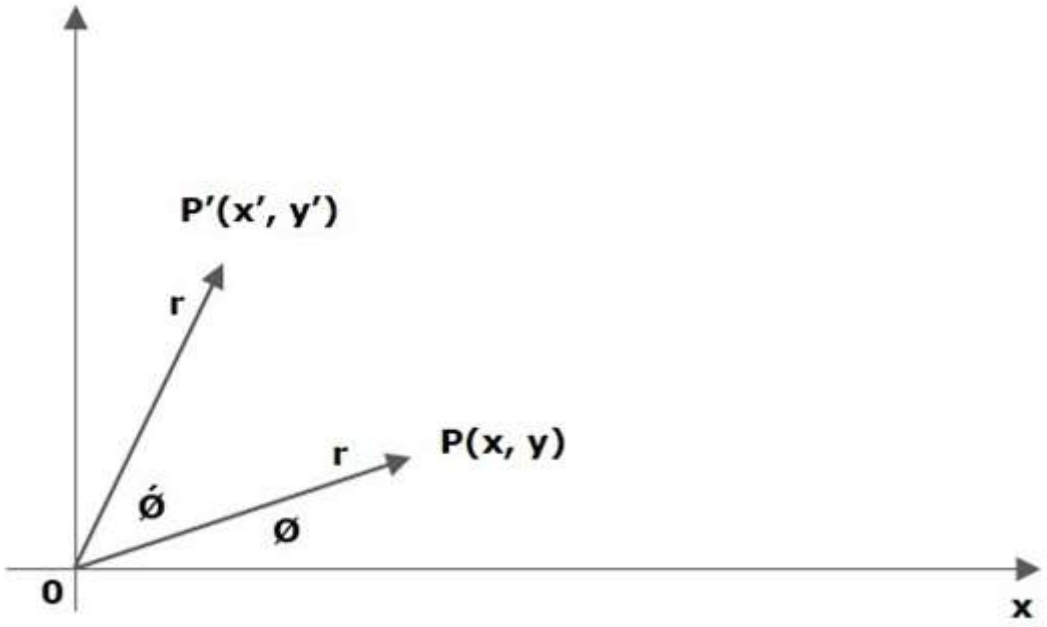
Transformation means changing some graphics into something else by applying rules. We can have various types of transformations such as **Translation**, **Scaling Up or Down**, **Rotation**, **Shearing**, etc.

Using JavaFX, you can apply transformations on nodes such as rotation, scaling and translation. All these transformations are represented by various classes and these belong to the package **javafx.scene.transform.**

S.No	Transformation & Description
1	**Rotation** In rotation, we rotate the object at a particular angle θ (theta) from its origin.
2	**Scaling** To change the size of an object, scaling transformation is used.
3	**Translation** Moves an object to a different position on the screen.
4	**Shearing** A transformation that slants the shape of an object is called the Shear Transformation.

Rotation

In rotation, we rotate the object at a particular angle **θ (theta)** from its origin. From the following figure, we can see that the **point P(X, Y)** is located at **angle φ** from the horizontal X coordinate with distance **r** from the origin.



Example

Following is the program which demonstrates the rotation transformation in JavaFX. In here, we are creating 2 rectangular nodes at the same location, with the same dimensions but with different colors (Blurywood and Blue). We are also applying rotation transformation on the rectangle with Blurywood color.

Save this code in a file with the name **RotationExample.java.**

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

public class RotationExample extends Application {

   @Override
   public void start(Stage stage) {

      //Drawing Rectangle1
      Rectangle rectangle1 = new Rectangle(150, 75, 200, 150);
```

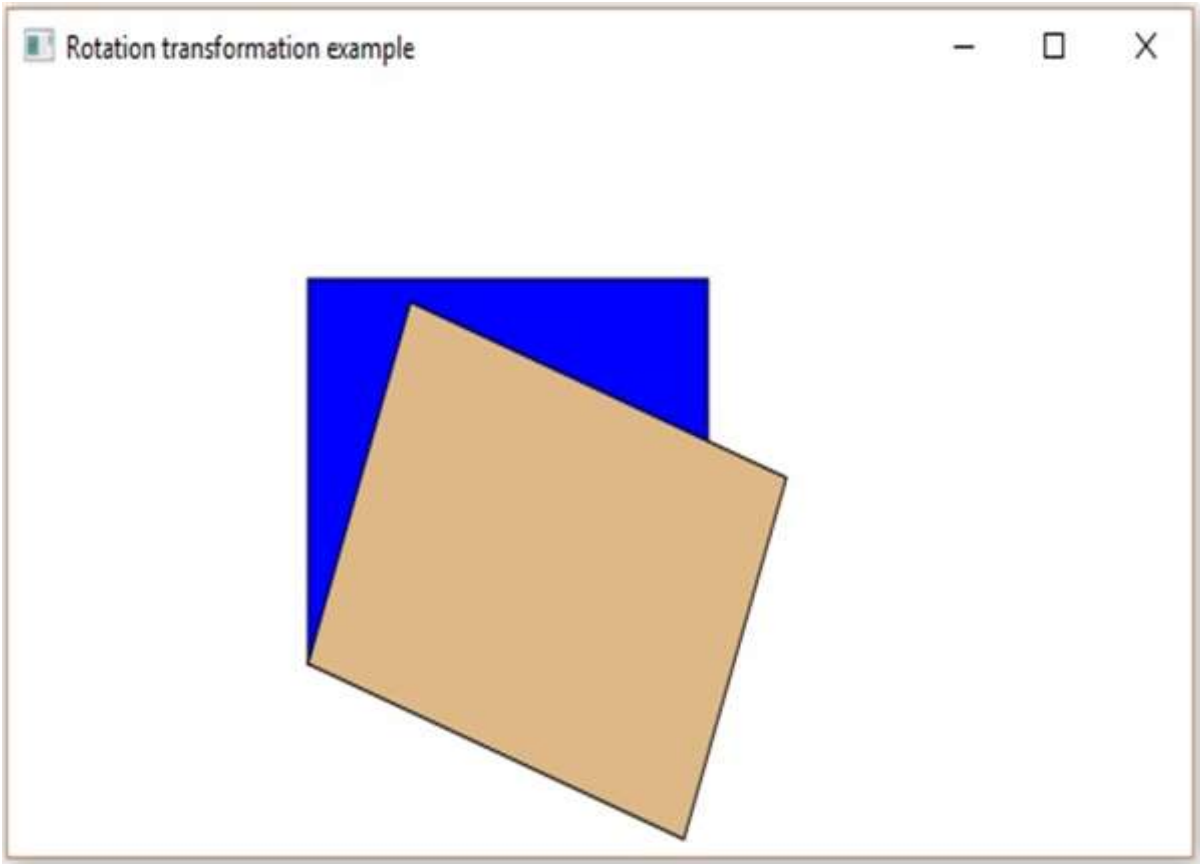
```
        rectangle1.setFill(Color.BLUE);
        rectangle1.setStroke(Color.BLACK);

        //Drawing Rectangle2
        Rectangle rectangle2 = new Rectangle(150, 75, 200, 150);
        //Setting the color of the rectangle
        rectangle2.setFill(Color.BURLYWOOD);
        //Setting the stroke color of the rectangle
        rectangle2.setStroke(Color.BLACK);

        //creating the rotation transformation
        Rotate rotate = new Rotate();
        //Setting the angle for the rotation
        rotate.setAngle(20);
        //Setting pivot points for the rotation
        rotate.setPivotX(150);
        rotate.setPivotY(225);



        //Adding the transformation to rectangle2
        rectangle2.getTransforms().addAll(rotate);

        //Creating a Group object
        Group root = new Group(rectangle1, rectangle2);

        //Creating a scene object
        Scene scene = new Scene(root, 600, 300);

        //Setting title to the Stage
        stage.setTitle("Rotation transformation example");

        //Adding scene to the satge
        stage.setScene(scene);

        //Displaying the contents of the stage
        stage.show();
```

```
    }

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

Compile and execute the saved java file from the command prompt using the following commands.

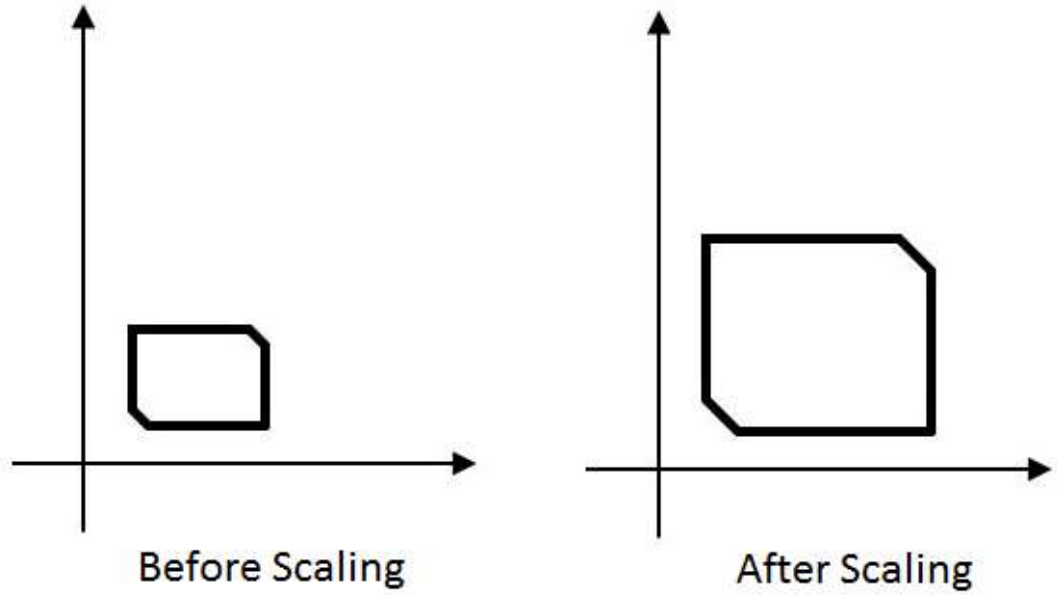
```
javac RotationExample.java
java RotationExample
```

On executing, the above program generates a javaFx window as shown below.



Scaling

To change the size of an object, scaling transformation is used. In the scaling process, you either expand or compress the dimensions of the object. Scaling can be achieved by multiplying the original coordinates of the object with the scaling factor to get the desired result.



Example

Following is the program which demonstrates scaling in JavaFX. Here, we are creating 2 circles (nodes) at the same location with the same dimensions, but with different colors (Blurywood and Blue). We are also applying scaling transformation on the circle with a blue color.

Save this code in a file with the name **ScalingExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.transform.Scale;
import javafx.stage.Stage;

public class ScalingExample extends Application {

   @Override
   public void start(Stage stage) {

      //Drawing Circle1
      Circle circle1 = new Circle(300, 135, 50);
```

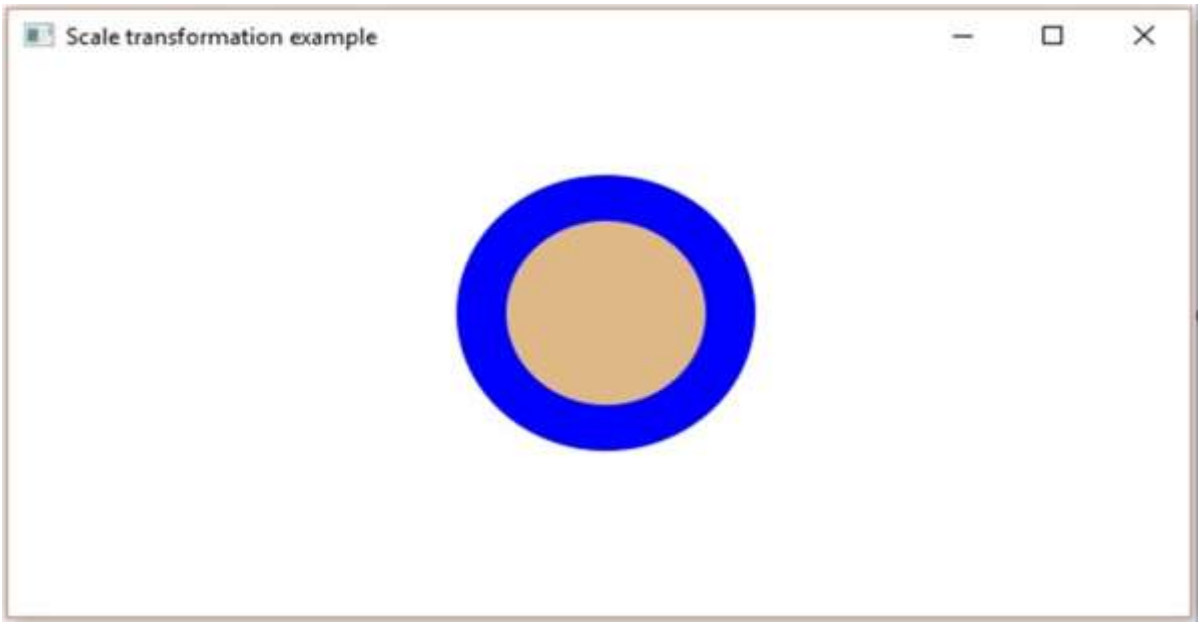
```
//Setting the color of the circle
circle1.setFill(Color.BLUE);

//Setting the stroke width of the circle
circle1.setStrokeWidth(20);

//Drawing Circle2
Circle circle2 = new Circle(300, 135, 50);
//Setting the color of the circle
circle2.setFill(Color.BURLYWOOD);
//Setting the stroke width of the circle
circle2.setStrokeWidth(20);

//Creating the scale transformation
Scale scale = new Scale();
//Setting the dimensions for the transformation
scale.setX(1.5);
scale.setY(1.5);
//Setting the pivot point for the transformation
scale.setPivotX(300);
scale.setPivotY(135);

//Adding the scale transformation to circle1
circle1.getTransforms().addAll(scale);

//Creating a Group object
Group root = new Group(circle1, circle2);

//Creating a scene object
Scene scene = new Scene(root, 600, 300);

//Setting title to the Stage
stage.setTitle("Scaling transformation example");
```

```
      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }

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

Compile and execute the saved java file from the command prompt using the following commands.

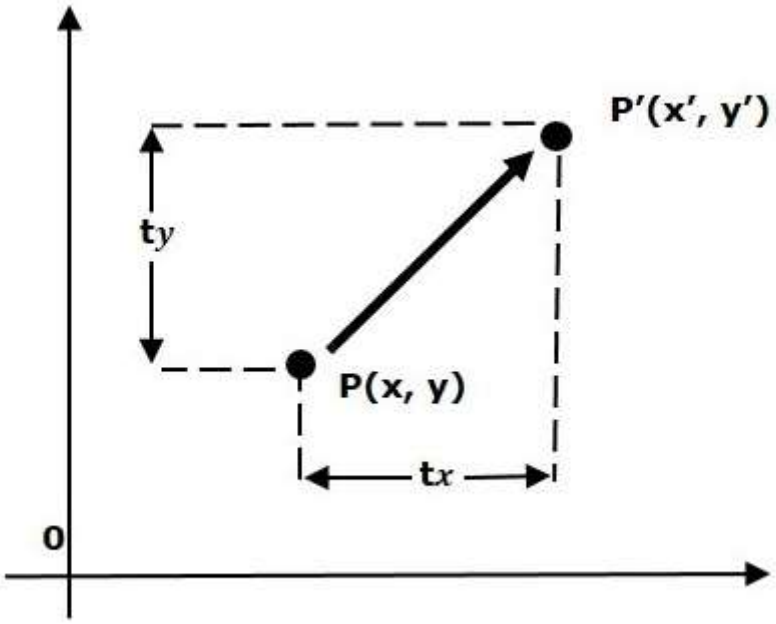
```
javac ScalingExample.java
java ScalingExample
```

On executing, the above program generates a JavaFX window as shown below.



Translation

A translation moves an object to a different position on the screen. You can translate a point in 2D by adding translation coordinate (t_x, t_y) to the original coordinate (X, Y) to get the new coordinate (X', Y').



Example

Following is the program which demonstrates translation in JavaFX. Here, we are creating 2 circles (nodes) at the same location with the same dimensions, but with different colors (Brown and Cadetblue). We are also applying translation on the circle with a **cadetblue** color.

Save this code in a file with the name **TranslationExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.transform.Translate;
import javafx.stage.Stage;

public class TranslationExample extends Application {
   @Override
   public void start(Stage stage) {
```

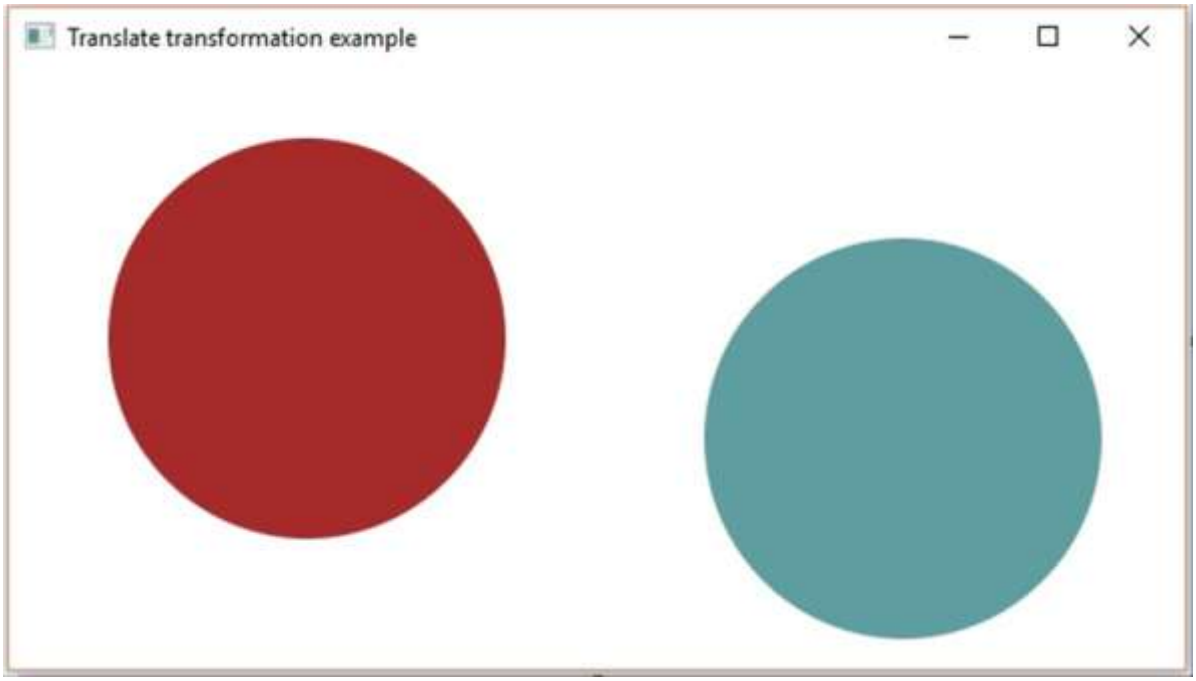
```
//Drawing Circle1
Circle circle = new Circle();
//Setting the position of the circle
circle.setCenterX(150.0f);
circle.setCenterY(135.0f);
//Setting the radius of the circle
circle.setRadius(100.0f);
//Setting the color of the circle
circle.setFill(Color.BROWN);
//Setting the stroke width of the circle
circle.setStrokeWidth(20);

//Drawing Circle2
Circle circle2 = new Circle();
//Setting the position of the circle
circle2.setCenterX(150.0f);
circle2.setCenterY(135.0f);
//Setting the radius of the circle
circle2.setRadius(100.0f);
//Setting the color of the circle
circle2.setFill(Color.CADETBLUE);
//Setting the stroke width of the circle
circle2.setStrokeWidth(20);

//Creating the translation transformation
Translate translate = new Translate();
//Setting the X,Y,Z coordinates to apply the translation
translate.setX(300);
translate.setY(50);
translate.setZ(100);

//Adding transformation to circle2
circle2.getTransforms().addAll(translate);

//Creating a Group object
```

```
      Group root = new Group(circle,circle2);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Translation transformation example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }
   public static void main(String args[]){
      launch(args);
   }
}
```

Compile and execute the saved java file from the command prompt using the following commands.

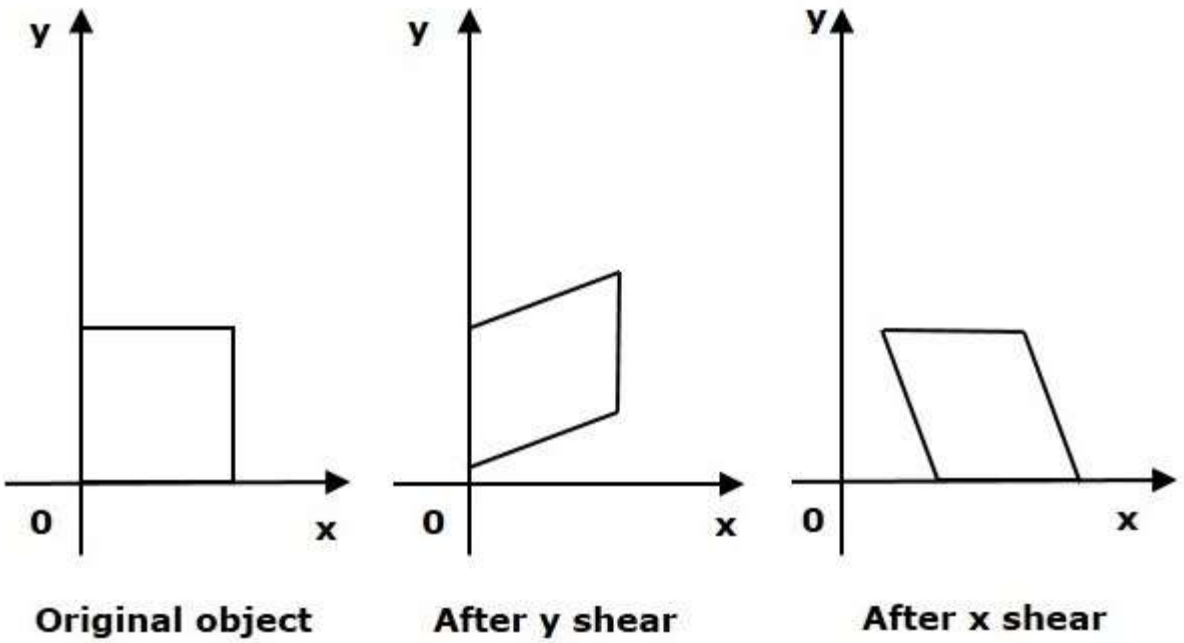
```
javac TranslationExample.java
java TranslationExample
```

On executing, the above program generates a JavaFX window as shown below.



Shearing

A transformation that slants the shape of an object is called the Shear Transformation. There are two shear transformations **X-Shear** and **Y-Shear**. One shifts the X coordinate values and the other shifts the Y coordinate values. However, in both the cases only one coordinate changes its coordinates and the other preserves its values. Shearing is also termed as **Skewing**.



Example

Following is the program which demonstrates shearing in JavaFX. Here, we are creating 2 polygons (nodes) at the same location, with the same dimensions, but with different colors (Blue and Transparent). We are also applying shearing on the transparent polygon.

Save this code in a file with the name **ShearingExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Polygon;
import javafx.scene.transform.Shear;
import javafx.stage.Stage;

public class ShearingExample extends Application {
   @Override
   public void start(Stage stage) {

      Polygon hexagon1 = new Polygon();
      //Adding coordinates to the hexagon
      hexagon1.getPoints().addAll(new Double[]{

         200.0, 50.0,
         400.0, 50.0,
         450.0, 150.0,
         400.0, 250.0,
         200.0, 250.0,
         150.0, 150.0,
          });
      //Setting the fill color for the hexagon
      hexagon1.setFill(Color.BLUE);
      hexagon1.setStroke(Color.BLACK);

      Polygon hexagon2 = new Polygon();
      //Adding coordinates to the hexagon
      hexagon2.getPoints().addAll(new Double[]{
         200.0, 50.0,
```

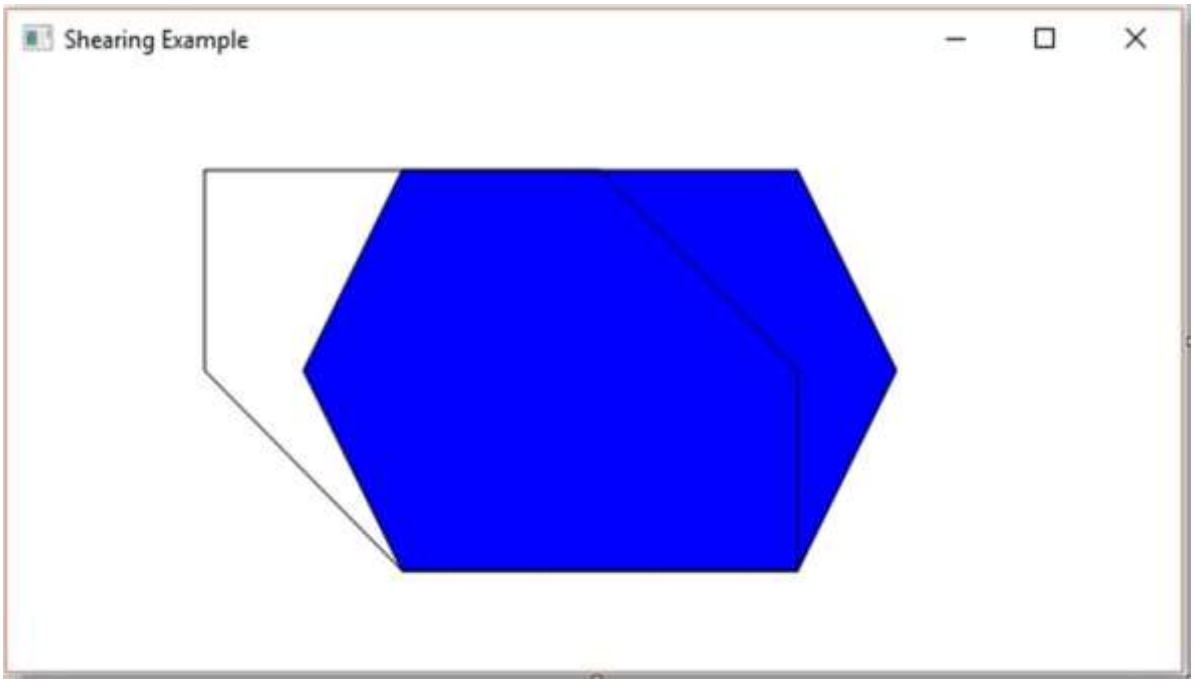
```
            400.0, 50.0,
            450.0, 150.0,
            400.0, 250.0,
            200.0, 250.0,
            150.0, 150.0,
             });
        //Setting the fill color for the hexagon
        hexagon2.setFill(Color.TRANSPARENT);
        hexagon2.setStroke(Color.BLACK);

        //Creating shear transformation
        Shear shear = new Shear();
        //Setting the pivot points
        shear.setPivotX(200);
        shear.setPivotY(250);
        //Setting the dimensions for the shear
        shear.setX(0.5);
        shear.setY(0.0);

        //Adding the transformation to the polygon

        hexagon2.getTransforms().addAll(shear);

        //Creating a Group object
        Group root = new Group(hexagon1, hexagon2);

        //Creating a scene object
        Scene scene = new Scene(root, 600, 300);

        //Setting title to the Stage
        stage.setTitle("Shearing Example ");

        //Adding scene to the satge
        stage.setScene(scene);

        //Displaying the contents of the stage
```

```
      stage.show();

   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac ShearingExample.java
java ShearingExample
```

On executing, the above program generates a JavaFX window as shown below.



Multiple Transformations

You can also apply multiple transformations on nodes in JavaFX. The following program is an example which performs **Rotation**, **Scaling** and **Translation** transformations on a rectangle simultaneously.

Save this code in a file with the name **MultipleTransformationsExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
```

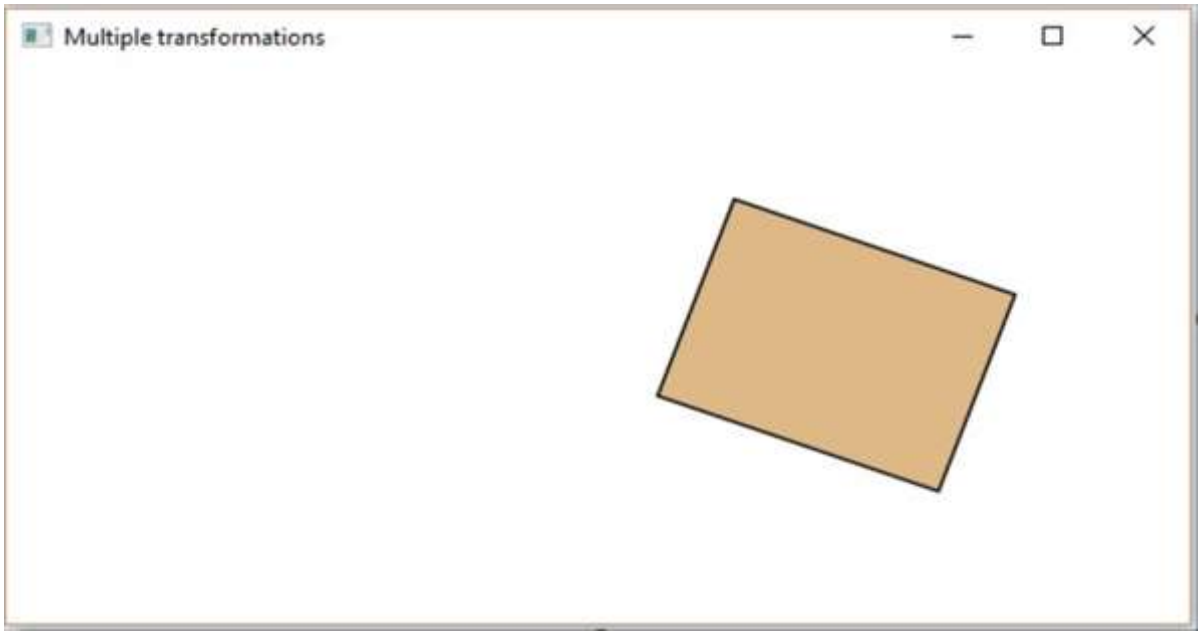
```
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.transform.Rotate;
import javafx.scene.transform.Scale;
import javafx.scene.transform.Translate;
import javafx.stage.Stage;

public class MultipleTransformationsExample extends Application {

   @Override
   public void start(Stage stage) {
      //Drawing a Rectangle
      Rectangle rectangle = new Rectangle(50, 50, 100, 75);
      //Setting the color of the rectangle
      rectangle.setFill(Color.BURLYWOOD);
      //Setting the stroke color of the rectangle
      rectangle.setStroke(Color.BLACK);

      //creating the rotation transformation
      Rotate rotate = new Rotate();
      //Setting the angle for the rotation
      rotate.setAngle(20);
      //Setting pivot points for the rotation
      rotate.setPivotX(150);
      rotate.setPivotY(225);

      //Creating the scale transformation
      Scale scale = new Scale();
      //Setting the dimensions for the transformation
      scale.setX(1.5);
      scale.setY(1.5);
      //Setting the pivot point for the transformation
      scale.setPivotX(300);
      scale.setPivotY(135);
```

```
      //Creating the translation transformation
      Translate translate = new Translate();
      //Setting the X,Y,Z coordinates to apply the translation
      translate.setX(250);
      translate.setY(0);
      translate.setZ(0);

      //Adding all the transformations to the rectangle
      rectangle.getTransforms().addAll(rotate, scale, translate);

      //Creating a Group object
      Group root = new Group(rectangle);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Multiple transformations");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac MultipleTransformationsExample.java
java MultipleTransformationsExample
```

On executing, the above program generates a JavaFX window as shown below.



Transformations on 3D Objects

You can also apply transformations on 3D objects. Following is an example which rotates and translates a 3-Dimensional box.

Save this code in a file with the name **Transformations3D.java**.

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.shape.Box;
import javafx.scene.transform.Rotate;
import javafx.scene.transform.Translate;
import javafx.stage.Stage;

public class RotationExample3D extends Application {

   @Override
   public void start(Stage stage) {
```

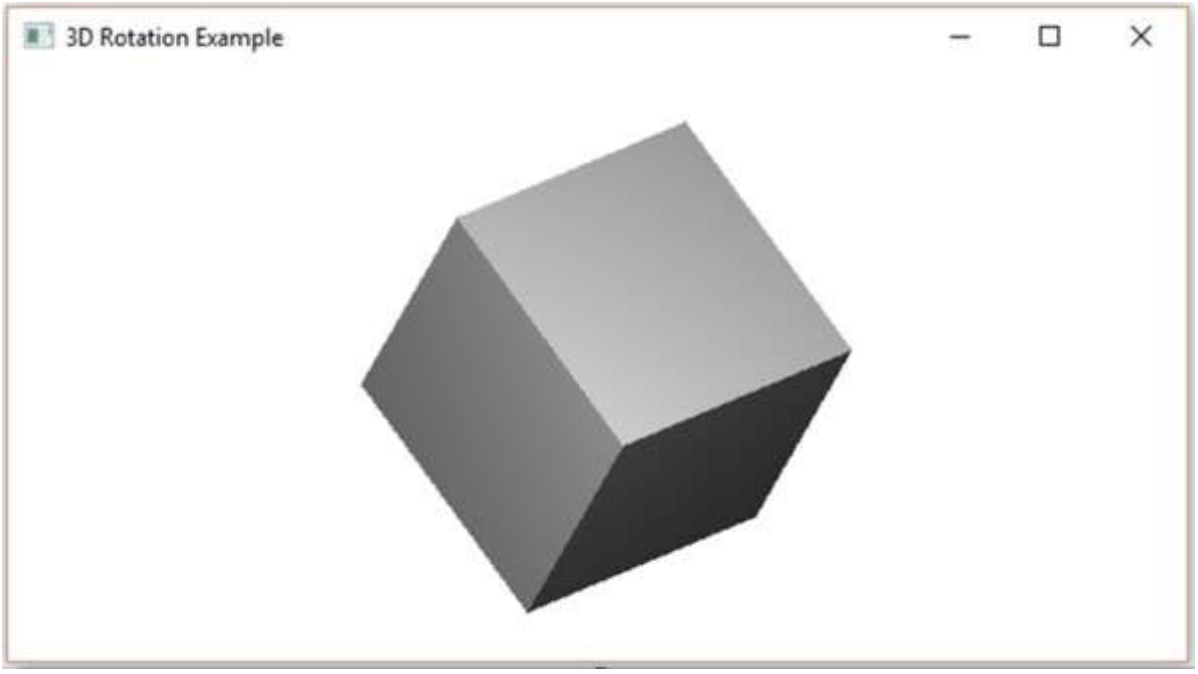
```
//Drawing a Box
Box box = new Box();

//Setting the properties of the Box
box.setWidth(150.0);
box.setHeight(150.0);
box.setDepth(150.0);

//Creating the translation transformation
Translate translate = new Translate();
translate.setX(400);
translate.setY(150);
translate.setZ(25);

Rotate rxBox = new Rotate(0, 0, 0, 0, Rotate.X_AXIS);
Rotate ryBox = new Rotate(0, 0, 0, 0, Rotate.Y_AXIS);
Rotate rzBox = new Rotate(0, 0, 0, 0, Rotate.Z_AXIS);
rxBox.setAngle(30);
ryBox.setAngle(50);
rzBox.setAngle(30);
box.getTransforms().addAll(translate,rxBox, ryBox, rzBox);

//Creating a Group object
Group root = new Group(box);

//Creating a scene object
Scene scene = new Scene(root, 600, 300);

//Setting title to the Stage
stage.setTitle("Drawing a cylinder");

//Adding scene to the satge
stage.setScene(scene);

//Displaying the contents of the stage
```

```
      stage.show();

   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac RotationExample3D.java
java RotationExample3D
```

On executing, the above program generates a JavaFX window as shown below.



9. JavaFX – Animations

In general, animating an object implies creating illusion of its motion by rapid display. In JavaFX, a node can be animated by changing its property over time. JavaFX provides a package named **javafx.animation.** This package contains classes that are used to animate the nodes. Animation is the base class of all these classes.

Using JavaFX, you can apply animations (transitions) such as **Fade Transition**, **Fill Transition**, **Rotate Transition**, **Scale Transition**, **Stroke Transition**, **Translate Transition**, **Path Transition**, **Sequential Transition**, **Pause Transition**, **Parallel Transition**, etc.

All these transitions are represented by individual classes in the package **javafx.animation**.

To apply a particular animation to a node, you have to follow the steps given below:

- Create a required node using respective class.
- Instantiate the respective transition (animation) class that is to be applied
- Set the properties of the transition and
- Finally play the transition using the **play()** method of the **Animation** class.

In this chapter, we are going to discuss examples of basic transitions (Rotation, Scaling, Translation).

Fade Transition

Following is the program which demonstrates Fade Transition in JavaFX. Save this code in a file with the name **FadeTransitionExample.java.**

```
import javafx.animation.FadeTransition;
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
import javafx.util.Duration;

public class FadeTransitionExample extends Application {
```

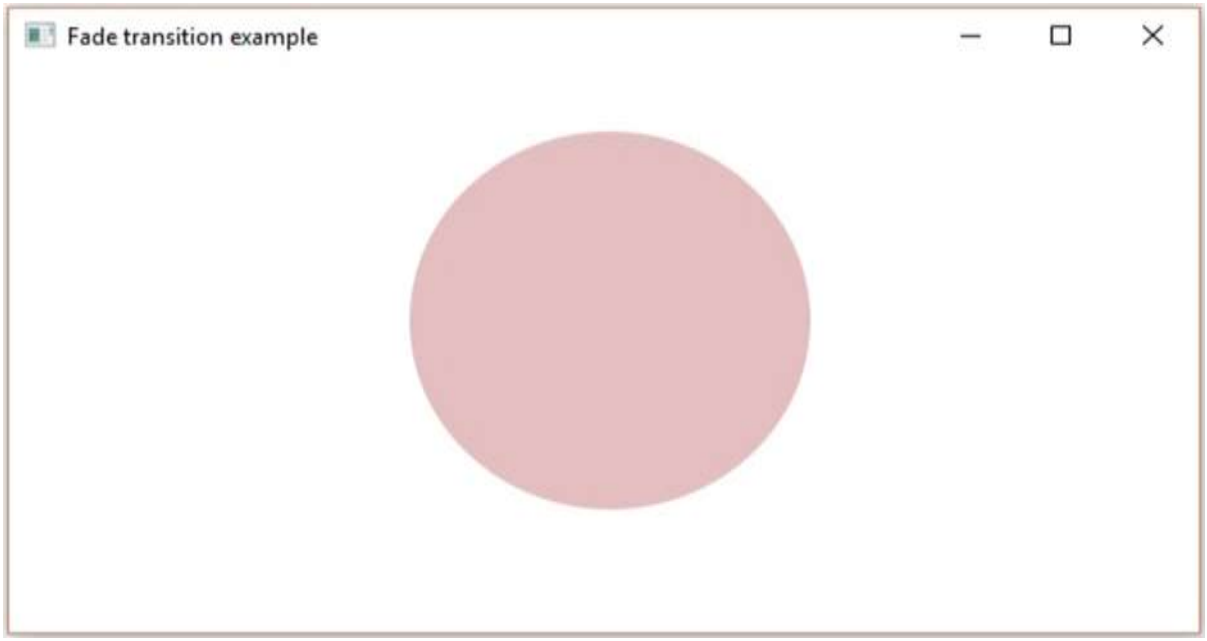
```
  @Override
   public void start(Stage stage) {
      //Drawing a Circle
      Circle circle = new Circle();
      //Setting the position of the circle
      circle.setCenterX(300.0f);

      circle.setCenterY(135.0f);
      //Setting the radius of the circle
      circle.setRadius(100.0f);
      //Setting the color of the circle
      circle.setFill(Color.BROWN);
      //Setting the stroke width of the circle
      circle.setStrokeWidth(20);

      //Creating the fade Transition
      FadeTransition fadeTransition = new
FadeTransition(Duration.millis(1000));
      //Setting the node for Transition
      fadeTransition.setNode(circle);
      //Setting the property fromValue of the transition (opacity)
      fadeTransition.setFromValue(1.0);
      //Setting the property toValue of the tansition (opacity)
      fadeTransition.setToValue(0.3);
      //Setting the cycle count for the transition
      fadeTransition.setCycleCount(50);
      //Setting auto reverse value to false
      fadeTransition.setAutoReverse(false);

     //Playing the animation
      fadeTransition.play();

      //Creating a Group object
      Group root = new Group(circle);

      //Creating a scene object
```

```
      Scene scene = new Scene(root, 600, 300);
      //Setting title to the Stage
      stage.setTitle("Fade transition example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac FadeTransitionExample.java
java FadeTransitionExample
```

On executing, the above program generates a JavaFX window as shown below.



Fill Transition

Following is the program which demonstrates Fill Transition in JavaFX. Save this code in a file with the name **FillTransitionExample.java**.

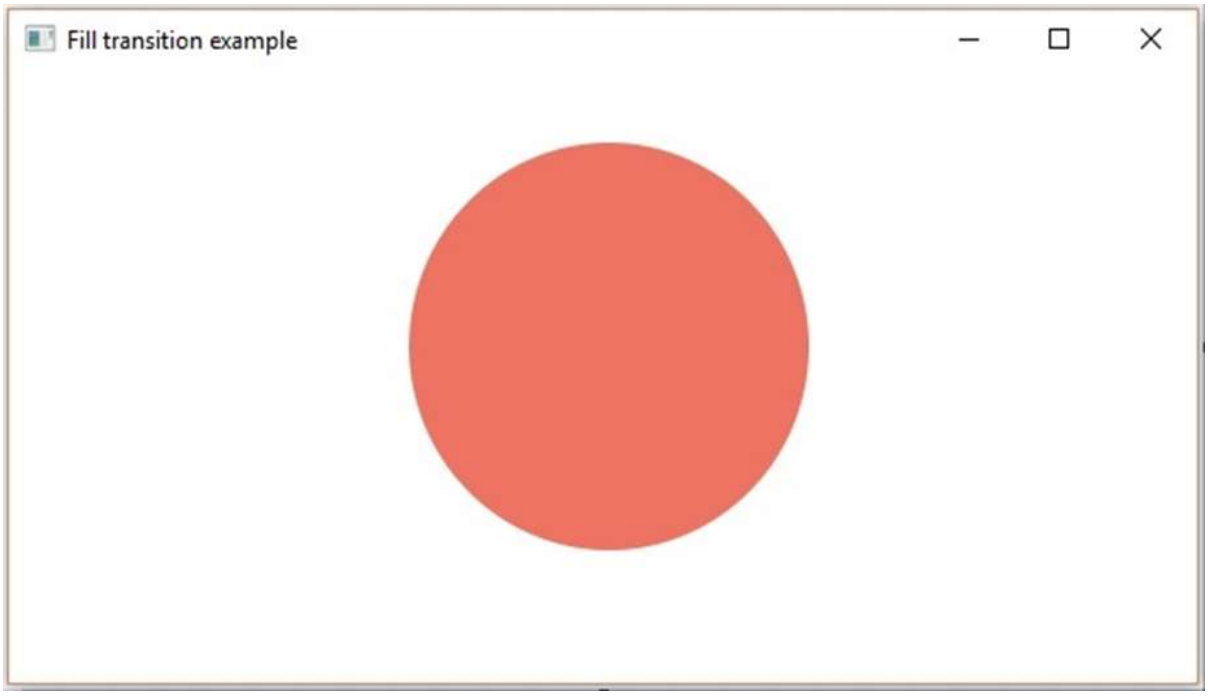
```
import javafx.animation.FillTransition;
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
import javafx.util.Duration;

public class FillTransitionExample extends Application {

  @Override
   public void start(Stage stage) {



      //Drawing a Circle
      Circle circle = new Circle();
      //Setting the position of the circle
      circle.setCenterX(300.0f);
      circle.setCenterY(135.0f);
      //Setting the radius of the circle
      circle.setRadius(100.0f);
      //Setting the color of the circle
      circle.setFill(Color.BROWN);
      //Setting the stroke width of the circle
      circle.setStrokeWidth(20);

      //Creating the fill Transition
      FillTransition fillTransition = new
FillTransition(Duration.millis(1000));
      //Setting the shape for Transition
      fillTransition.setShape(circle);
      //Setting the from value of the transition (color)
```

```
      fillTransition.setFromValue(Color.BLUEVIOLET);

      //Setting the toValue of the tansition (color)
      fillTransition.setToValue(Color.CORAL);
      //Setting the cycle count for the transition
      fillTransition.setCycleCount(50);
      //Setting auto reverse value to false
      fillTransition.setAutoReverse(false);

      //Playing the animation
      fillTransition.play();

      //Creating a Group object
      Group root = new Group(circle);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);



      //Setting title to the Stage
      stage.setTitle("Fill transition example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac FillTransitionExample.java
java FillTransitionExample
```

On executing, the above program generates a JavaFX window as shown below.



Rotate Transition

Following is the program which demonstrates Rotate Transition in JavaFX. Save this code in a file with the name **RotateTransitionExample.java**.

```
import javafx.animation.RotateTransition;
import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Polygon;
import javafx.stage.Stage;
import javafx.util.Duration;

public class RotateTransitionExample extends Application {
   @Override
   public void start(Stage stage) {
```

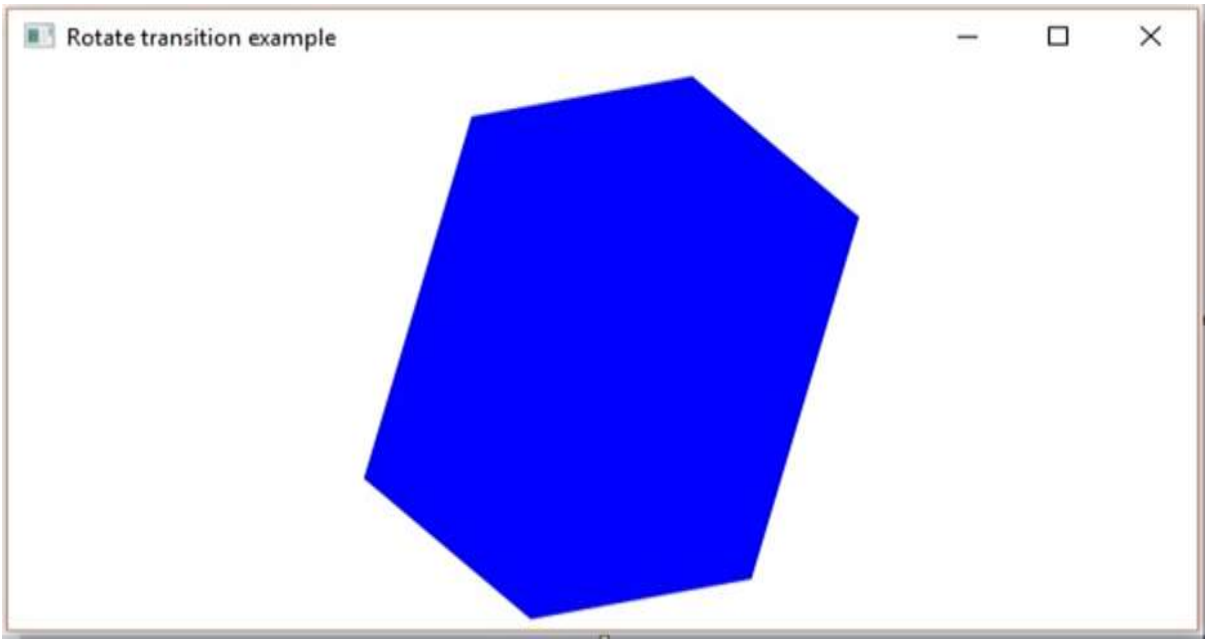
```
//Creating a hexagon
Polygon hexagon = new Polygon();
//Adding coordinates to the hexagon
hexagon.getPoints().addAll(new Double[]{
   200.0, 50.0,
   400.0, 50.0,
   450.0, 150.0,
   400.0, 250.0,
   200.0, 250.0,
   150.0, 150.0,
    });
//Setting the fill color for the hexagon
hexagon.setFill(Color.BLUE);

//Creating a rotate transition
RotateTransition rotateTransition = new RotateTransition();
//Setting the duration for the transition
rotateTransition.setDuration(Duration.millis(1000));
//Setting the node for the transition
rotateTransition.setNode(hexagon);
//Setting the angle of the rotation
rotateTransition.setByAngle(360);
//Setting the cycle count for the transition
rotateTransition.setCycleCount(50);
//Setting auto reverse value to false
rotateTransition.setAutoReverse(false);
//Playing the animation
rotateTransition.play();

//Creating a Group object

Group root = new Group(hexagon);

//Creating a scene object
Scene scene = new Scene(root, 600, 300);
```

```
      //Setting title to the Stage
      stage.setTitle("Rotate transition example ");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac RotateTransitionExample.java
java RotateTransitionExample
```

On executing, the above program generates a JavaFX window as shown below.



Scale Transition

Following is the program which demonstrates Scale Transition in JavaFX. Save this code in a file with the name **ScaleTransitionExample.java**.

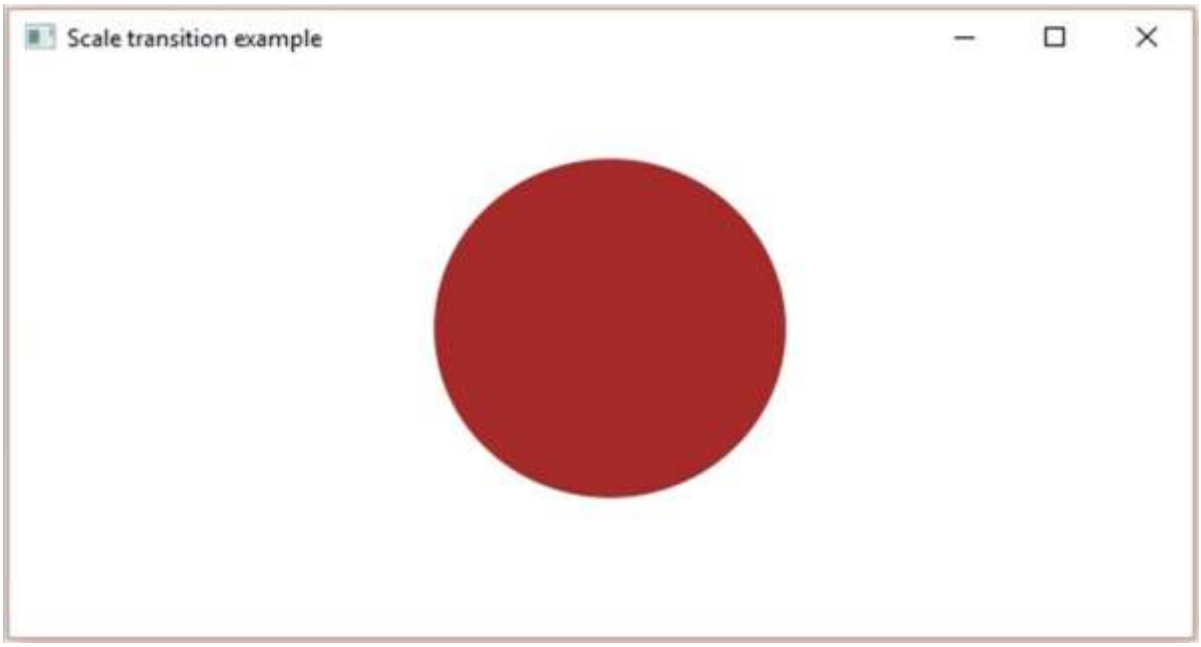
```
import javafx.animation.ScaleTransition;
import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
import javafx.util.Duration;

public class ScaleTransitionExample extends Application {

   @Override
   public void start(Stage stage) {

      //Drawing a Circle
      Circle circle = new Circle();
      //Setting the position of the circle
      circle.setCenterX(300.0f);
      circle.setCenterY(135.0f);
      //Setting the radius of the circle
      circle.setRadius(50.0f);
      //Setting the color of the circle
      circle.setFill(Color.BROWN);
      //Setting the stroke width of the circle
      circle.setStrokeWidth(20);

      //Creating scale Transition
      ScaleTransition scaleTransition = new ScaleTransition();
      //Setting the duration for the transition
      scaleTransition.setDuration(Duration.millis(1000));
      //Setting the node for the transition
      scaleTransition.setNode(circle);
```

```
      //Setting the dimensions for scaling
      scaleTransition.setByY(1.5);
      scaleTransition.setByX(1.5);
      //Setting the cycle count for the translaton
      scaleTransition.setCycleCount(50);
      //Setting auto reverse value to true
      scaleTransition.setAutoReverse(false);
      //Playing the animation
      scaleTransition.play();

      //Creating a Group object
      Group root = new Group(circle);

      //Creating a scene object

      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Scale transition example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac ScaleTransitionExample.java
java ScaleTransitionExample
```


On executing, the above program generates a JavaFX window as shown below.



Stroke Transition

Following is the program which demonstrates Stoke Transition in JavaFX. Save this code in a file with the name **StrokeTransitionExample.java**.

```
import javafx.animation.StrokeTransition;
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
import javafx.util.Duration;

public class StrokeTransitionExample extends Application {

   @Override
   public void start(Stage stage) {

      //Drawing a Circle
      Circle circle = new Circle();
      //Setting the position of the circle
```

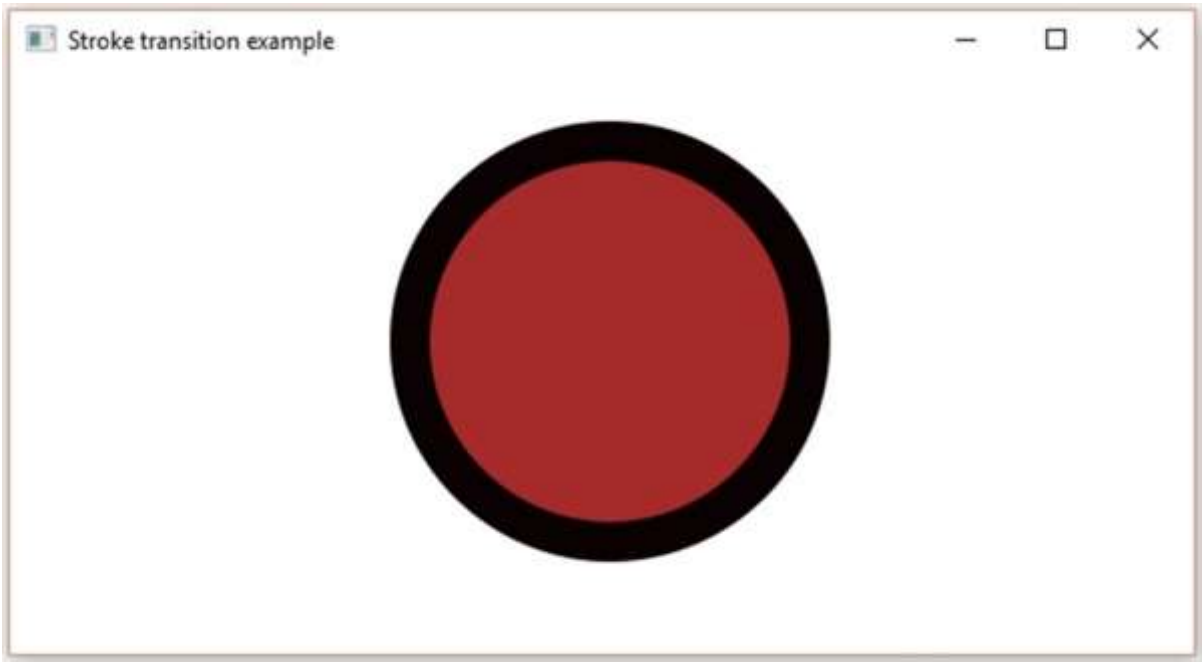
```
circle.setCenterX(300.0f);
circle.setCenterY(135.0f);
//Setting the radius of the circle
circle.setRadius(100.0f);
//Setting the color of the circle
circle.setFill(Color.BROWN);
//Setting the stroke width of the circle
circle.setStrokeWidth(20);

//creating stroke transition
StrokeTransition strokeTransition = new StrokeTransition();
//Setting the duration fo the transition
strokeTransition.setDuration(Duration.millis(1000));
//Setting the shape for the transition
strokeTransition.setShape(circle);
//Setting the fromValue property of the transition (color)
strokeTransition.setFromValue(Color.BLACK);
//Setting the toValue property of the transition (color)
strokeTransition.setToValue(Color.BROWN);

//Setting the cycle count for the transition
strokeTransition.setCycleCount(50);
//Setting auto reverse value to false
strokeTransition.setAutoReverse(false);
//Playing the animation
strokeTransition.play();

//Creating a Group object
Group root = new Group(circle);

//Creating a scene object
Scene scene = new Scene(root, 600, 300);

//Setting title to the Stage
stage.setTitle("Stroke transition example");
```

```
      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac StrokeTransitionExample.java
java StrokeTransitionExample
```

On executing, the above program generates a JavaFX window as shown below.



Translate Transition

Following is the program which demonstrates Translate Transition in JavaFX. Save this code in a file with the name **TranslateTransitionExample.java**.

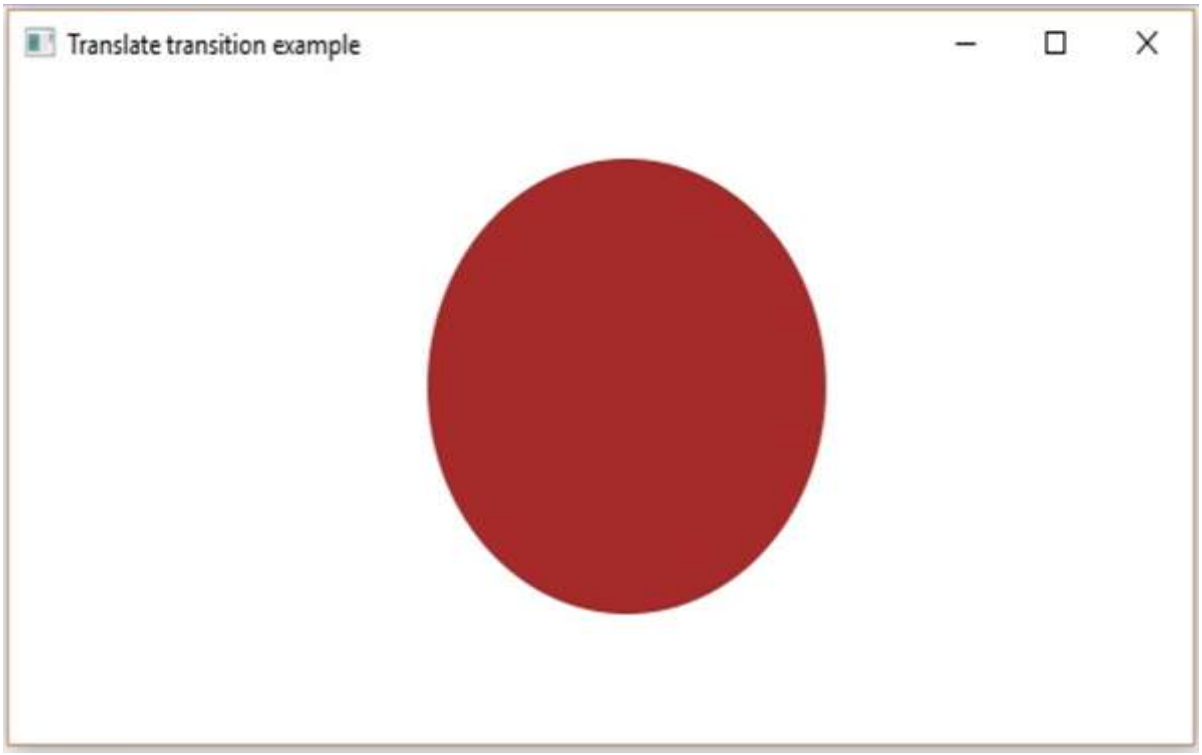
```
import javafx.animation.TranslateTransition;
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
import javafx.util.Duration;

public class TranslateTransitionExample extends Application {
   @Override
   public void start(Stage stage) {


     //Drawing a Circle
      Circle circle = new Circle();
      //Setting the position of the circle
      circle.setCenterX(150.0f);
      circle.setCenterY(135.0f);
      //Setting the radius of the circle
      circle.setRadius(100.0f);
      //Setting the color of the circle
      circle.setFill(Color.BROWN);
      //Setting the stroke width of the circle
      circle.setStrokeWidth(20);

      //Creating Translate Transition
      TranslateTransition translateTransition = new TranslateTransition();
      //Setting the duration of the transition
      translateTransition.setDuration(Duration.millis(1000));
      //Setting the node for the transition
      translateTransition.setNode(circle);
      //Setting the value of th transition along the x axis.
      translateTransition.setByX(300);
      //Setting the cycle count for the transition
      translateTransition.setCycleCount(50);
```

```
      //Setting auto reverse value to false
      translateTransition.setAutoReverse(false);
      //Playing the animation
      translateTransition.play();

      //Creating a Group object
      Group root = new Group(circle);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Translate transition example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac TranslateTransitionExample.java
java TranslateTransitionExample
```

On executing, the above program generates a JavaFX window as shown below.



Sequential Transition

Following is the program which demonstrates Sequential Transition in JavaFX. Save this code in a file with the name **SequentialTransitionExample.java**.

```
import javafx.animation.PathTransition;
import javafx.animation.ScaleTransition;
import javafx.animation.SequentialTransition;
import javafx.animation.TranslateTransition;
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.CubicCurveTo;
import javafx.scene.shape.MoveTo;
import javafx.scene.shape.Path;
import javafx.stage.Stage;

import javafx.util.Duration;

public class SequentialTransitionExample extends Application {
```

```
@Override
public void start(Stage stage) {

   //Drawing a Circle
   Circle circle = new Circle();
   //Setting the position of the circle
   circle.setCenterX(150.0f);
   circle.setCenterY(135.0f);
   //Setting the radius of the circle
   circle.setRadius(100.0f);
   //Setting the color of the circle
   circle.setFill(Color.BROWN);
   //Setting the stroke width of the circle
   circle.setStrokeWidth(20);

   //Instantiating the path class
   Path path = new Path();
   //Creating the MovrTo path element
   MoveTo moveTo = new MoveTo(100, 150);
   //Creating the Cubic curve path element
   CubicCurveTo cubicCurveTo = new CubicCurveTo(400, 40, 175, 250, 500, 150);
   //Adding the path elements to Observable list of the Path class
   path.getElements().add(moveTo);
   path.getElements().add(cubicCurveTo);

   //Creating path Transition
   PathTransition pathTransition = new PathTransition();
   //Setting the duration for the transition
   pathTransition.setDuration(Duration.millis(1000));
   //Setting the node for the transition
   pathTransition.setNode(circle);
   //Setting the path for the transition

   pathTransition.setPath(path);
   //Setting the orientation of the path
```

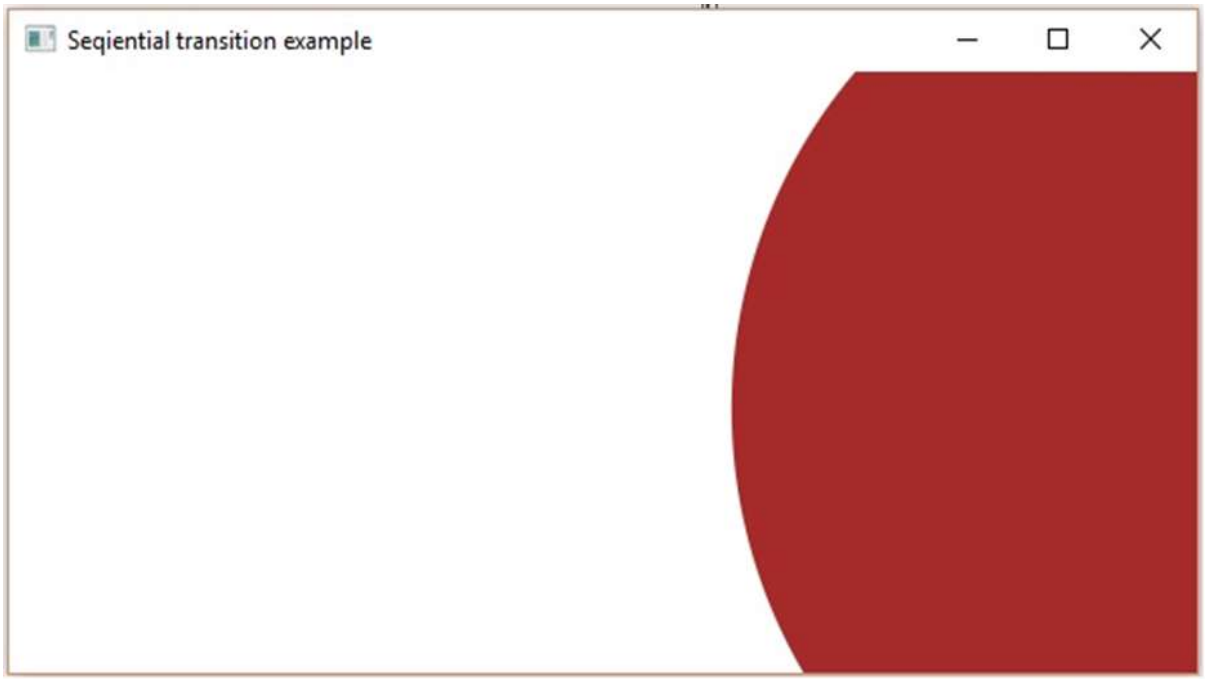
```
pathTransition.setOrientation(PathTransition.OrientationType.ORTHOGONAL_TO_TAN
GENT);
      //Setting the cycle count for the transition
      pathTransition.setCycleCount(5);
      //Setting auto reverse value to false
      pathTransition.setAutoReverse(false);
      //Playing the animation
      pathTransition.play();

      //Creating Translate Transition
      TranslateTransition translateTransition = new TranslateTransition();
      //Setting the duration for the transition
      pathTransition.setDuration(Duration.millis(1000));
      //Setting the node for the transition
      pathTransition.setNode(circle);
      //Setting the length of the transition along x axis
      translateTransition.setByX(300);
      //Setting the cycle count for the stroke
      translateTransition.setCycleCount(5);
      //Setting auto reverse value to false
      translateTransition.setAutoReverse(false);

      //Applying scale Transition to the circle
      ScaleTransition scaleTransition = new ScaleTransition();
      //Setting the duration for the transition
      pathTransition.setDuration(Duration.millis(1000));
      //Setting the node for the transition
      pathTransition.setNode(circle);
      //Setting the dimensions for scaling
      scaleTransition.setByY(1.5);
      scaleTransition.setByX(1.5);

      //Setting the cycle count for the translaton
      scaleTransition.setCycleCount(5);
      //Setting auto reverse value to false
      scaleTransition.setAutoReverse(false);
```

```
      //Applying Sequential Translation to the circle
      SequentialTransition sequentialTransition = new
SequentialTransition(circle, pathTransition, translateTransition,
scaleTransition );
      //Playing the animation
      sequentialTransition.play();

      //Creating a Group object
      Group root = new Group(circle);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);
      //Setting title to the Stage
      stage.setTitle("Seqiential transition example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac SequentialTransitionExample.java
java SequentialTransitionExample
```

On executing, the above program generates a JavaFX window as shown below.



Parallel Transition

Following is the program which demonstrates Parallel Transition in JavaFX. Save this code in a file with the name **parallelTransitionExample.java**.

```
import javafx.animation.ParallelTransition;
import javafx.animation.PathTransition;
import javafx.animation.ScaleTransition;
import javafx.animation.TranslateTransition;
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;

import javafx.scene.shape.CubicCurveTo;
import javafx.scene.shape.MoveTo;
import javafx.scene.shape.Path;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.util.Duration;

public class parallelTransitionExample extends Application {
```

```
   @Override
   public void start(Stage stage) {

      //Drawing a Rectangle
      Rectangle rectangle = new Rectangle();
      //Setting the position of the rectangle
      rectangle.setX(75.0f);
      rectangle.setY(75.0f);
      //Setting the width of the rectangle
      rectangle.setWidth(100.0f);
      //Setting the height of the rectangle
      rectangle.setHeight(100.0f);
      //setting the color of the rectangle
      rectangle.setFill(Color.BLUEVIOLET);

      //Instantiating the path class
      Path path = new Path();
      //Creating the MoveTo path element
      MoveTo moveTo = new MoveTo(100, 150);
      //Creating the Cubic curve path element
      CubicCurveTo cubicCurveTo = new CubicCurveTo(400, 40, 175, 250, 500,
150);
      //Adding the path elements to Observable list of the Path class
      path.getElements().add(moveTo);
      path.getElements().add(cubicCurveTo);

      //Creating Path Transition

      PathTransition pathTransition = new PathTransition();
      //Setting the duration of the transition
      pathTransition.setDuration(Duration.millis(1000));
      //Setting the node for the transition
      pathTransition.setNode(rectangle);
      //Setting the path for the transition

      pathTransition.setPath(path);
```

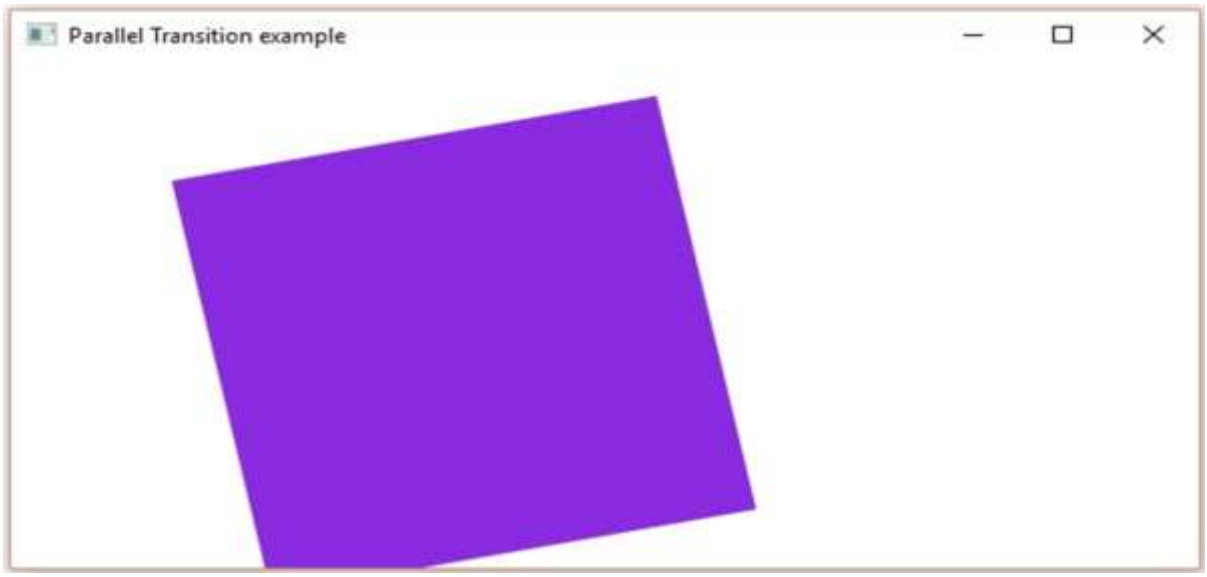
```
      //Setting the orientation of the path

pathTransition.setOrientation(PathTransition.OrientationType.ORTHOGONAL_TO_TAN
GENT);
      //Setting the cycle count for the transition
      pathTransition.setCycleCount(5);
      //Setting auto reverse value to false
      pathTransition.setAutoReverse(false);
      //Playing the animation
      pathTransition.play();

      //Ctreating Translate Transition
      TranslateTransition translateTransition = new TranslateTransition();
      //Setting the duration for the transition
      translateTransition.setDuration(Duration.millis(1000));
      //Setting the node for the transition
      translateTransition.setNode(rectangle);
      //Setting the axis and length of the transition
      translateTransition.setByX(300);
      //Setting the cycle count ofthe transition
      translateTransition.setCycleCount(5);
      //Setting auto reverse value to false
      translateTransition.setAutoReverse(false);

      //Creating scale Transition
      ScaleTransition scaleTransition = new ScaleTransition();
      //Setting the duration for the transition

      translateTransition.setDuration(Duration.millis(1000));
      //Setting the node for the transition
      translateTransition.setNode(rectangle);
      //Setting the dimensions for scaling
      scaleTransition.setByY(1.5);
      scaleTransition.setByX(1.5);
      //Setting the cycle count for the translaton

      scaleTransition.setCycleCount(5);
```

```
      //Setting auto reverse value to true
      scaleTransition.setAutoReverse(false);

      //Applying parallel Translation to the circle
      ParallelTransition parallelTransition = new
ParallelTransition(rectangle, pathTransition, translateTransition,
scaleTransition );
      //Playing the animation
      parallelTransition.play();

      //Creating a Group object
      Group root = new Group(rectangle);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Parallel Transition example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac parallelTransitionExample.java
java parallelTransitionExample
```

On executing, the above program generates a JavaFX window as shown below.



Pause Transition

Following is the program which demonstrates Pause Transition in JavaFX. Save this code in a file with the name **PauseTransitionExample.java**.

```
import javafx.animation.PauseTransition;
import javafx.animation.ScaleTransition;

import javafx.animation.SequentialTransition;
import javafx.animation.TranslateTransition;
import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;

import javafx.stage.Stage;
import javafx.util.Duration;

public class PauseTransitionExample extends Application {

   @Override
   public void start(Stage stage) {
```

```
//Drawing a Circle
Circle circle = new Circle();
//Setting the position of the circle
circle.setCenterX(150.0f);
circle.setCenterY(135.0f);
//Setting the radius of the circle
circle.setRadius(50.0f);
//Setting the color of the circle
circle.setFill(Color.BROWN);
//Setting the stroke width of the circle
circle.setStrokeWidth(20);

//Creating a Pause Transition
PauseTransition pauseTransition = new PauseTransition();
//Setting the duration for the transition
pauseTransition.setDuration(Duration.millis(1000));

//Creating Translate Transition
TranslateTransition translateTransition = new TranslateTransition();

//Setting the duration for the transition
translateTransition.setDuration(Duration.millis(1000));
//Setting the node of the transition
translateTransition.setNode(circle);
//Setting the value of the transition along the x axis
translateTransition.setByX(300);
//Setting the cycle count for the stroke
translateTransition.setCycleCount(5);

//Setting auto reverse value to true
translateTransition.setAutoReverse(false);

//Creating scale Transition
ScaleTransition scaleTransition = new ScaleTransition();
//Setting the duration for the transition
```

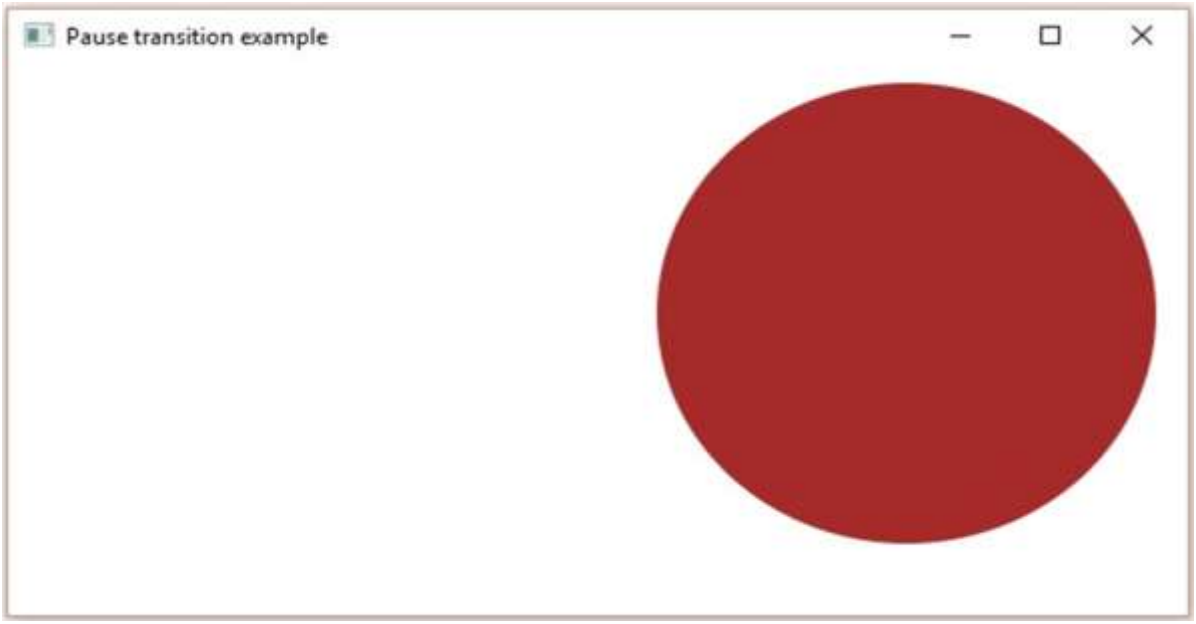
```
      scaleTransition.setDuration(Duration.millis(1000));
      //Setting the node for the transition
      scaleTransition.setNode(circle);
      //Setting the dimensions for scaling
      scaleTransition.setByY(1.5);
      scaleTransition.setByX(1.5);
      //Setting the cycle count for the translaton
      scaleTransition.setCycleCount(5);
      //Setting auto reverse value to true
      scaleTransition.setAutoReverse(false);

      //Applying Sequential transition to the circle
      SequentialTransition sequentialTransition = new
SequentialTransition(circle, translateTransition, pauseTransition,
scaleTransition );
      //Playing the animation
      sequentialTransition.play();

      //Creating a Group object
      Group root = new Group(circle);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Pause transition example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }

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

```
    }
}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac PauseTransitionExample.java
java PauseTransitionExample
```

On executing, the above program generates a JavaFX window as shown below.



PathTransition

Following is the program which demonstrates Path Transistion in JavaFX. Save this code in a file with the name **PathTransitionExample.java.**

```
import javafx.animation.PathTransition;
import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.CubicCurveTo;
import javafx.scene.shape.MoveTo;
import javafx.scene.shape.Path;
import javafx.stage.Stage;
```

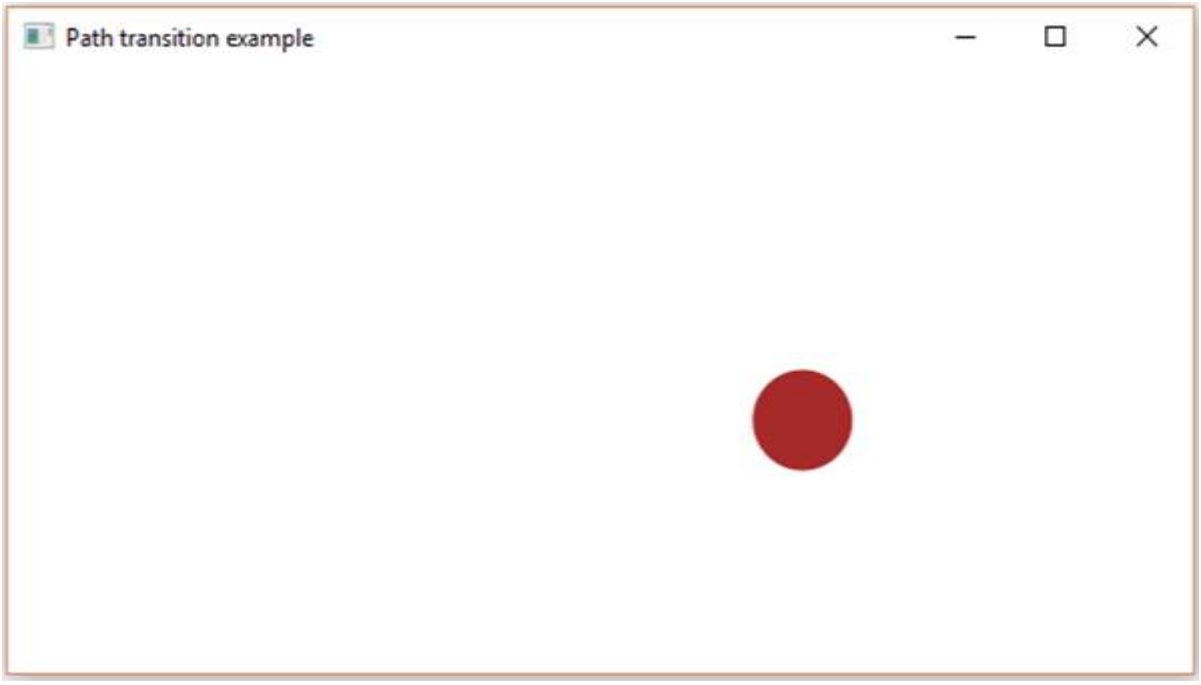
```
import javafx.util.Duration;



public class PathTransitionExample extends Application {
   @Override
   public void start(Stage stage) {

      //Drawing a Circle
      Circle circle = new Circle();
      //Setting the position of the circle
      circle.setCenterX(300.0f);
      circle.setCenterY(135.0f);
      //Setting the radius of the circle
      circle.setRadius(25.0f);
      //Setting the color of the circle
      circle.setFill(Color.BROWN);
      //Setting the stroke width of the circle
      circle.setStrokeWidth(20);

      //Instantiating the path class

      Path path = new Path();
      //Creating the MovrTo path element
      MoveTo moveTo = new MoveTo(100, 150);
      //Creating the Cubic curve path element
      CubicCurveTo cubicCurveTo = new CubicCurveTo(400, 40, 175, 250, 500,
150);
      //Adding the path elements to Observable list of the Path class
      path.getElements().add(moveTo);
      path.getElements().add(cubicCurveTo);

      //Creating a path transition
      PathTransition pathTransition = new PathTransition();
      //Setting the duration of the path transition
      pathTransition.setDuration(Duration.millis(1000));
      //Setting the node for the transition
```

```
      pathTransition.setNode(circle);
      //Setting the path
      pathTransition.setPath(path);

      //Setting the orientation of the path

pathTransition.setOrientation(PathTransition.OrientationType.ORTHOGONAL_TO_TAN
GENT);
      //Setting the cycle count for the transition
      pathTransition.setCycleCount(50);
      //Setting auto reverse value to false
      pathTransition.setAutoReverse(false);

      //Playing the animation
      pathTransition.play();

      //Creating a Group object
      Group root = new Group(circle);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Path transition example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac PathTransitionExample.java
java PathTransitionExample
```

On executing, the above program generates a JavaFX window as shown below.



Example – 2

Following is an example which transforms a circle along a complex path. Save this code in a file with name **PathTransitionExample2.java**

```
import javafx.animation.PathTransition;
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.LineTo;
import javafx.scene.shape.MoveTo;
import javafx.scene.shape.Path;

import javafx.stage.Stage;
import javafx.util.Duration;
```

```
public class PathTransitionExample2 extends Application {



   @Override
   public void start(Stage stage) {

      //Drawing a Circle
      Circle circle = new Circle();
      //Setting the position of the circle
      circle.setCenterX(300.0f);
      circle.setCenterY(135.0f);
      //Setting the radius of the circle
      circle.setRadius(25.0f);
      //Setting the color of the circle
      circle.setFill(Color.BROWN);
      //Setting the stroke width of the circle
      circle.setStrokeWidth(20);

      //Creating a Path
      Path path = new Path();
      //Moving to the starting point
      MoveTo moveTo = new MoveTo(108, 71);
      //Creating 1st line
      LineTo line1 = new LineTo(321, 161);
      //Creating 2nd line
      LineTo line2 = new LineTo(126,232);
      //Creating 3rd line
      LineTo line3 = new LineTo(232,52);
      //Creating 4th line
      LineTo line4 = new LineTo(269, 250);
      //Creating 5th line
      LineTo line5 = new LineTo(108, 71);
      //Adding all the elements to the path

      path.getElements().add(moveTo);
      path.getElements().addAll(line1, line2, line3, line4, line5);
```

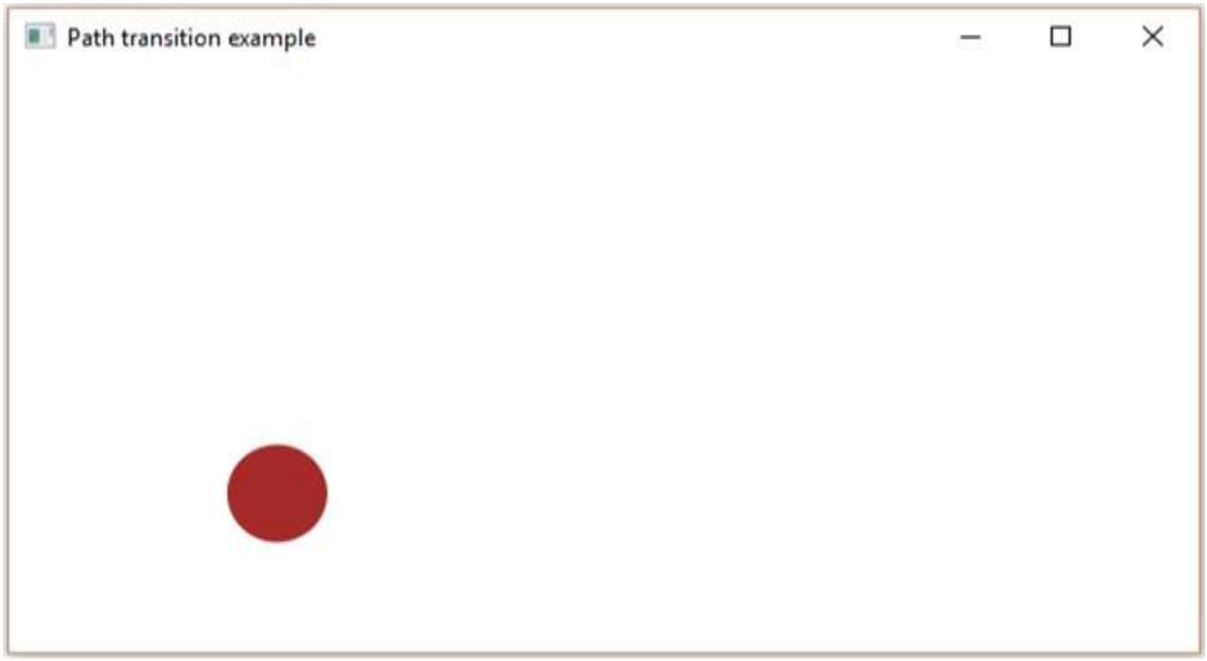
```
      //Creating the path transition
      PathTransition pathTransition = new PathTransition();
      //Setting the duration of the transition
      pathTransition.setDuration(Duration.millis(1000));
      //Setting the node for the transition
      pathTransition.setNode(circle);
      //Setting the path for the transition
      pathTransition.setPath(path);
      //Setting the orientation of the path

pathTransition.setOrientation(PathTransition.OrientationType.ORTHOGONAL_TO_TAN
GENT);
      //Setting the cycle count for the transition
      pathTransition.setCycleCount(50);
      //Setting auto reverse value to true
      pathTransition.setAutoReverse(false);
      //Playing the animation
      pathTransition.play();

      //Creating a Group object
      Group root = new Group(circle);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Path transition example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }
```

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac PathTransitionExample2.java
java PathTransitionExample2
```

On executing, the above program generates a JavaFX window as shown below.



10. JavaFX – Colors

To apply colors to an application, JavaFX provides various classes in the package **javafx.scene.paint** package. This package contains an abstract class named Paint and it is the base class of all the classes that are used to apply colors.

Using these classes, you can apply colors in the following patterns –

- **Uniform:** In this pattern, color is applied uniformly throughout node.
- **Image Pattern:** This lets you to fill the region of the node with an image pattern.
- **Gradient:** In this pattern, the color applied to the node varies from one point to the other. It has two kinds of gradients namely **Linear Gradient** and **Radial Gradient**.

All those node classes to which you can apply color such as **Shape**, **Text** (including Scene), have methods named **setFill()** and **setStroke()**. These will help to set the color values of the nodes and their strokes respectively.

These methods accept an object of type Paint. Therefore, to create either of these type of images, you need to instantiate these classes and pass the object as a parameter to these methods.

Applying Color to the Nodes

To set uniform color pattern to the nodes, you need to pass an object of the class color to the **setFill()**, **setStroke()** methods as follows –

```
//Setting color to the text
Color color = new Color.BEIGE
text.setFill(color);
//Setting color to the stroke
Color color = new Color.DARKSLATEBLUE
circle.setStroke(color);
```

In the above code block, we are using the static variables of the color class to create a color object.

In the same way, you can also use the RGB values or HSB standard of coloring or web hash codes of colors as shown below –

```
//creating color object by passing RGB values
Color c = Color.rgb(0,0,255);
//creating color object by passing HSB values
```

```
Color c = Color.hsb(270,1.0,1.0);
//creating color object by passing the hash code for web
Color c = Color.web("0x0000FF",1.0);
```

Example

Following is an example which demonstrates, how to apply color to the nodes in JavaFX. Here, we are creating a circle and text nodes and applying colors to them.

Save this code in a file with the name **ColorExample.java**.

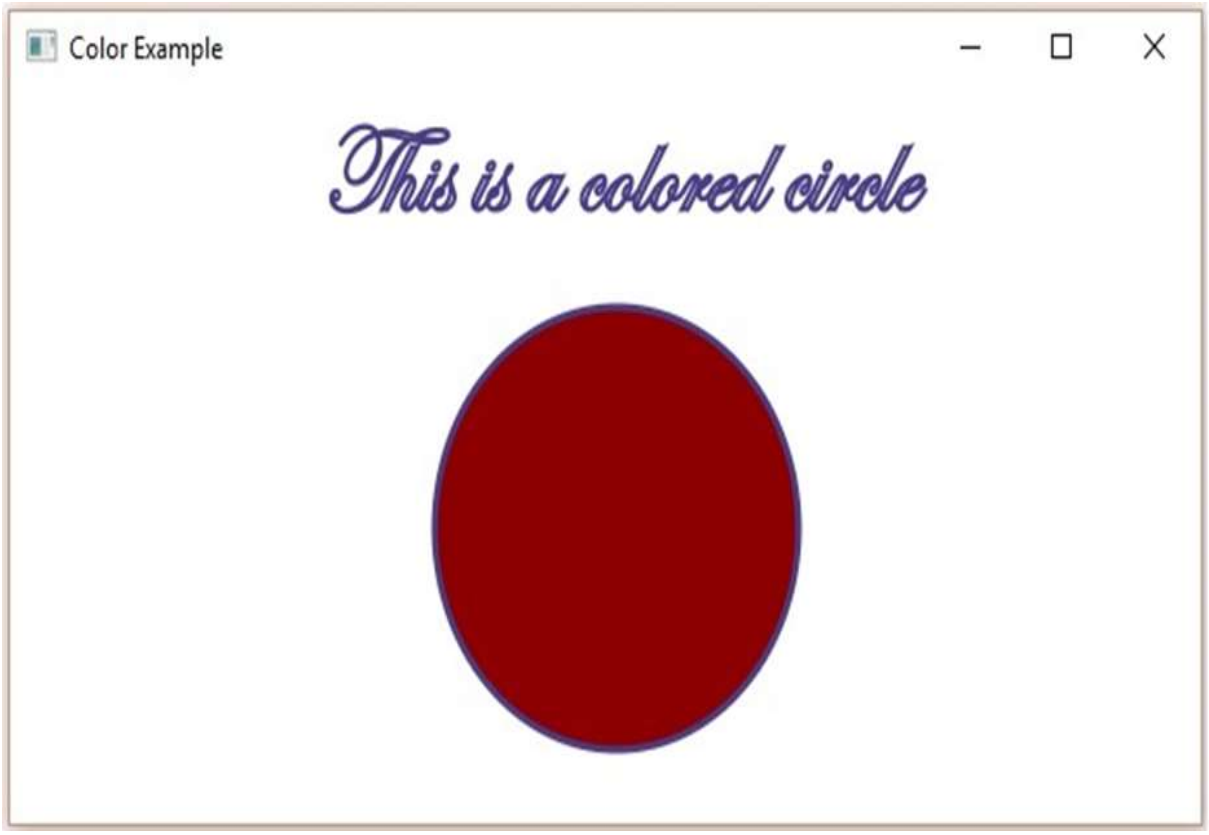
```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.scene.shape.Circle;
import javafx.scene.text.Font;
import javafx.scene.text.Text;

public class ColorExample extends Application {

   @Override
   public void start(Stage stage) {

      //Drawing a Circle
      Circle circle = new Circle();
      //Setting the properties of the circle
      circle.setCenterX(300.0f);
      circle.setCenterY(180.0f);
      circle.setRadius(90.0f);

      //Setting color to the circle
      circle.setFill(Color.DARKRED);
      //Setting the stroke width
      circle.setStrokeWidth(3);
      //Setting color to the stroke
      circle.setStroke(Color.DARKSLATEBLUE);
```

```
      //Drawing a text
      Text text = new Text("This is a colored circle");
      //Setting the font of the text
      text.setFont(Font.font("Edwardian Script ITC", 50));
      //Setting the position of the text
      text.setX(155);
      text.setY(50);

      //Setting color to the text
      text.setFill(Color.BEIGE);
      text.setStrokeWidth(2);
      text.setStroke(Color.DARKSLATEBLUE);

      //Creating a Group object
      Group root = new Group(circle, text);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Color Example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }

  public static void main(String args[]){

      launch(args);
   }
}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
Javac ColorExample.java

java ColorExample
```

On executing, the above program generates a JavaFX window as follows –



Applying Image Pattern to the Nodes

To apply an image pattern to the nodes, instantiate the **ImagePattern** class and pass its object to the **setFill()**, **setStroke()** methods.

The constructor of this class accepts six parameters namely –

- **image:** The object of the image using which you want to create the pattern.
- **x and y**: Double variables representing the (x, y) coordinates of origin of the anchor rectangle.
- **height and width:** Double variables representing the height and width of the image that is used to create a pattern.
- **isProportional:** This is a Boolean Variable; on setting this property to **true**, the start and end locations are set to be proportional.

```
ImagePattern radialGradient = new ImagePattern(dots, 20, 20, 40, 40, false);
```

Example

Following is an example which demonstrates how to apply image pattern to the nodes in JavaFX. Here, we are creating a circle and a text node and applying an image pattern to them

Save this code in a file with name **ImagePatternExample.java.**

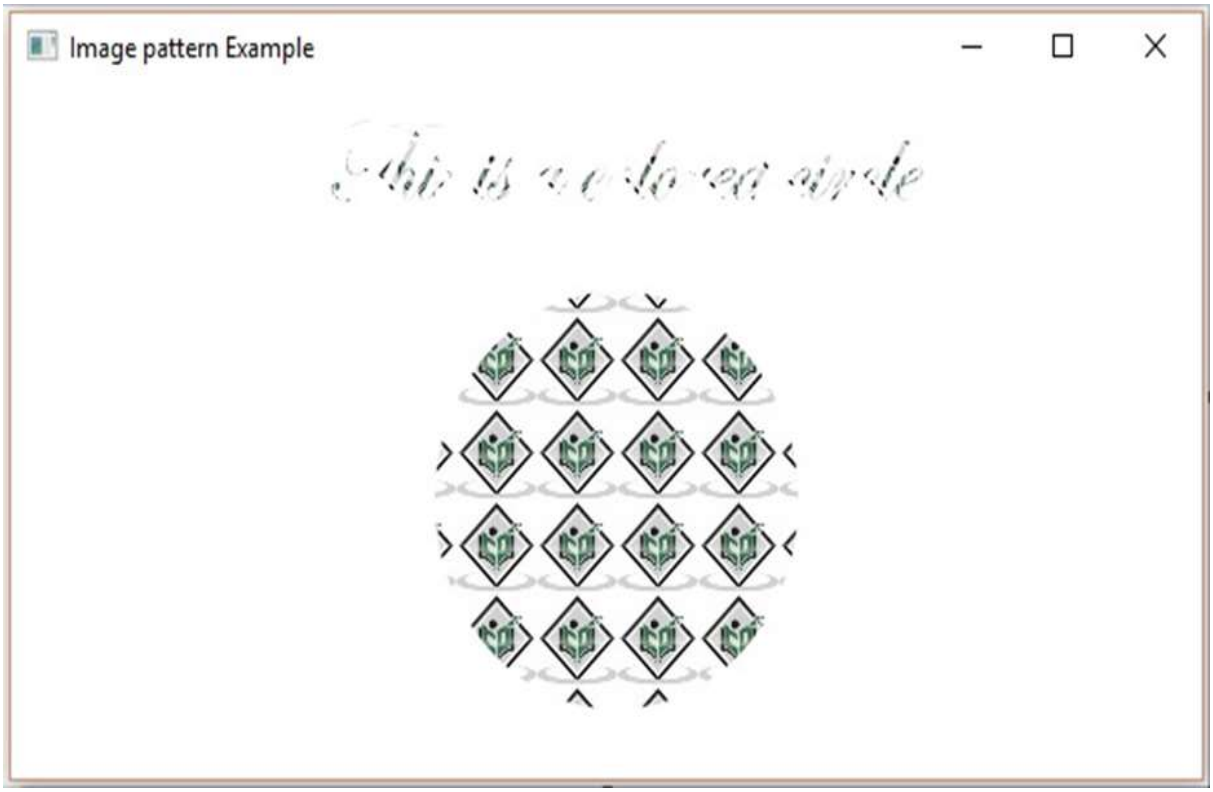
```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.paint.Color;
import javafx.scene.paint.ImagePattern;
import javafx.scene.paint.Stop;
import javafx.stage.Stage;
import javafx.scene.shape.Circle;
import javafx.scene.text.Font;
import javafx.scene.text.Text;

public class ImagePatternExample extends Application {

   @Override
   public void start(Stage stage) {

      //Drawing a Circle
      Circle circle = new Circle();
      //Setting the properties of the circle
      circle.setCenterX(300.0f);
      circle.setCenterY(180.0f);
      circle.setRadius(90.0f);

      //Drawing a text
      Text text = new Text("This is a colored circle");
      //Setting the font of the text
      text.setFont(Font.font("Edwardian Script ITC", 50));
      //Setting the position of the text
```

```
      text.setX(155);
      text.setY(50);

      //Setting the image pattern
      String link = "https://encrypted-tbn1.gstatic.com"
              + "/images?q=tbn:ANd9GcRQub4GvEezKMsiIf67U"
              + "rOxSzQuQ9zl5ysnjRn87VOC8tAdgmAJjcwZ2qM";
      Image image = new Image(link);
      ImagePattern radialGradient = new ImagePattern(image, 20, 20, 40, 40,
false);

      //Setting the linear gradient to the circle and text
      circle.setFill(radialGradient);
      text.setFill(radialGradient);

      //Creating a Group object
      Group root = new Group(circle, text);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Image pattern Example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
Javac ImagePatternExample.java
java ImagePatternExample
```

On executing, the above program generates a JavaFX window as follows –



Applying Linear Gradient Pattern

To apply a Linear Gradient Pattern to the nodes, instantiate the **LinearGradent** class and pass its object to the **setFill(), setStroke()** methods.

The constructor of this class accepts five parameters namely –

- **startX, startY:** These double properties represent the x and y coordinates of the starting point of the gradient.
- **endX, endY**: These double properties represent the x and y coordinates of the ending point of the gradient.
- **cycleMethod:** This argument defines how the regions outside the color gradient bounds, defined by the starting and ending points, should be filled.
- **proportional:** This is a Boolean Variable; on setting this property to **true**, the start and end locations are set to a proportion.
- **Stops:** This argument defines the color-stop points along the gradient line.

```
      //Setting the linear gradient
      Stop[] stops = new Stop[] {
          new Stop(0, Color.DARKSLATEBLUE),
          new Stop(1, Color.DARKRED)};

      LinearGradient linearGradient = new LinearGradient(0, 0, 1, 0, true,
CycleMethod.NO_CYCLE, stops);
```

Example

Following is an example which demonstrates how to apply a gradient pattern to the nodes in JavaFX. Here, we are creating a circle and a text nodes and applying linear gradient pattern to them.

Save this code in a file with name **LinearGradientExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.paint.CycleMethod;
import javafx.scene.paint.LinearGradient;
import javafx.scene.paint.Stop;
import javafx.stage.Stage;
import javafx.scene.shape.Circle;
import javafx.scene.text.Font;
import javafx.scene.text.Text;

public class LinearGradientExample extends Application {
   @Override
   public void start(Stage stage) {
      //Drawing a Circle
      Circle circle = new Circle();
      //Setting the properties of the circle
      circle.setCenterX(300.0f);

      circle.setCenterY(180.0f);
      circle.setRadius(90.0f);
```

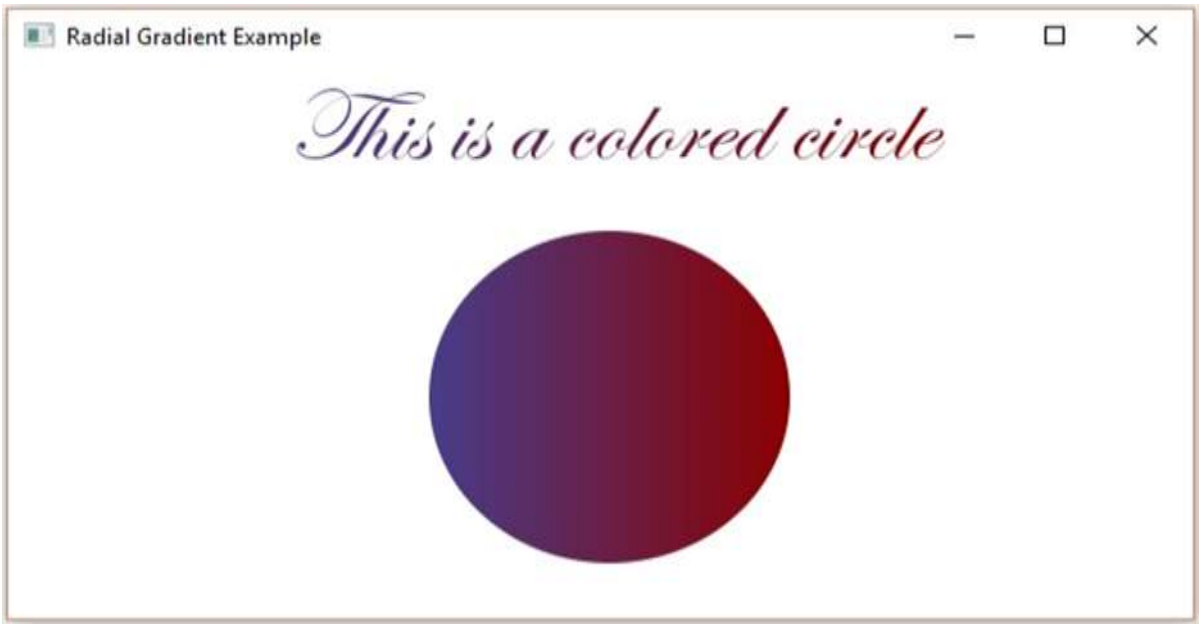
```
      //Drawing a text
      Text text = new Text("This is a colored circle");
      //Setting the font of the text
      text.setFont(Font.font("Edwardian Script ITC", 55));
      //Setting the position of the text
      text.setX(140);
      text.setY(50);

      //Settiing the linear gradient
      Stop[] stops = new Stop[] {
          new Stop(0, Color.DARKSLATEBLUE),
          new Stop(1, Color.DARKRED)};

      LinearGradient linearGradient = new LinearGradient(0, 0, 1, 0, true,
CycleMethod.NO_CYCLE, stops);

      //Setting the linear gradient to the circle and text
      circle.setFill(linearGradient);
      text.setFill(linearGradient);

      //Creating a Group object
      Group root = new Group(circle, text);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Radial Gradient Example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }
```

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

Compile and execute the saved java file from the command prompt using the following commands.

```
Javac LinearGradientExample.java
java LinearGradientExample
```

On executing, the above program generates a JavaFX window as follows –



Applying Radial Gradient Pattern

To apply a Linear Gradient Pattern to the nodes, instantiate the **GradientPattern** class and pass its object to the **setFill()**, **setStroke()** methods.

The constructor of this class accepts a few parameters, some of which are –

- **startX, startY:** These double properties represent the x and y coordinates of the starting point of the gradient.
- **endX, endY**: These double properties represent the x and y coordinates of the ending point of the gradient.
- **cycleMethod:** This argument defines how the regions outside the color gradient bounds are defined by the starting and ending points and how they should be filled.

- **proportional:** This is a Boolean Variable; on setting this property to **true** the start and end locations are set to a proportion.
- **Stops:** This argument defines the color-stop points along the gradient line.

```
//Setting the radial gradient
        Stop[] stops = new Stop[] {
          new Stop(0.0, Color.WHITE),
          new Stop(0.3, Color.RED),
          new Stop(1.0, Color.DARKRED)
        };

RadialGradient radialGradient = new RadialGradient(0, 0, 300, 178, 60, false,
CycleMethod.NO_CYCLE, stops);
```

Example

Following is an example which demonstrates how to apply a radial gradient pattern to the nodes in JavaFX. Here, we are creating a circle and a text nodes and applying gradient pattern to them.

Save this code in a file with the name **RadialGradientExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.paint.CycleMethod;
import javafx.scene.paint.RadialGradient;

import javafx.scene.paint.Stop;
import javafx.stage.Stage;
import javafx.scene.shape.Circle;
import javafx.scene.text.Font;
import javafx.scene.text.Text;



public class ColorRadialGradient extends Application {

    @Override
    public void start(Stage stage) {
```

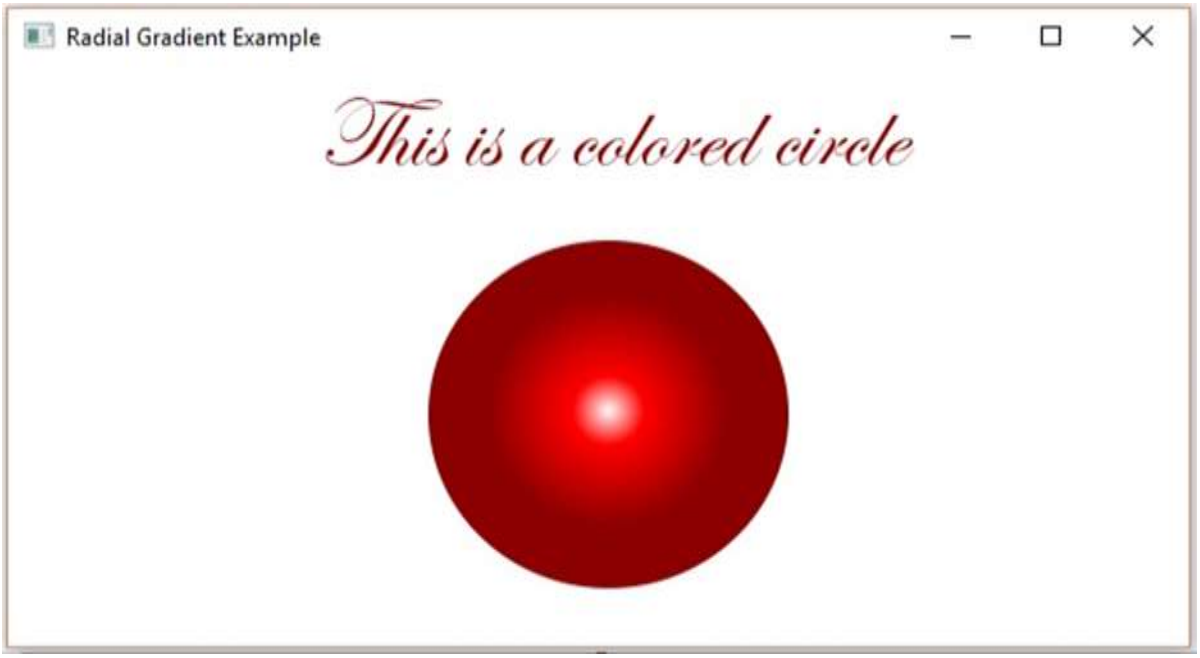
```
        //Drawing a Circle
        Circle circle = new Circle();
        //Setting the properties of the circle
        circle.setCenterX(300.0f);
        circle.setCenterY(180.0f);
        circle.setRadius(90.0f);

        //Drawing a text
        Text text = new Text("This is a colored circle");
        //Setting the font of the text
        text.setFont(Font.font("Edwardian Script ITC", 50));
        //Setting the position of the text
        text.setX(155);
        text.setY(50);

        //Settiing the radial gradient
        Stop[] stops = new Stop[] {
          new Stop(0.0, Color.WHITE),
          new Stop(0.3, Color.RED),
          new Stop(1.0, Color.DARKRED)
        };

        RadialGradient radialGradient = new RadialGradient(0, 0, 300, 178, 60,
false, CycleMethod.NO_CYCLE, stops);

        //Setting the linear gradient to the circle and text
        circle.setFill(radialGradient);
        text.setFill(radialGradient);

        //Creating a Group object
        Group root = new Group(circle, text);

        //Creating a scene object
        Scene scene = new Scene(root, 600, 300);
```

```
        //Setting title to the Stage
        stage.setTitle("Radial Gradient Example");

        //Adding scene to the satge
        stage.setScene(scene);

        //Displaying the contents of the stage
        stage.show();
    }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
Javac RadialGradientExample.java
java RadialGradientExample
```

On executing, the above program generates a JavaFX window as follows –



11. JavaFX – Images

You can load and modify images using the classes provided by JavaFX in the package **javafx.scene.image.** JavaFX supports the image formats like **Bmp**, **Gif**, **Jpeg**, **Png**.

This chapter teaches you how to load images in to JavaFX, how to project an image in multiple views and how to alter the pixels of an image.

Loading an Image

You can load an image in JavaFX by instantiating the class named **Image** of the package **javafx.scene.image**.

To the constructor of the class, you have to pass either of the following –

- An **InputStream** object of the image to be loaded or,
- A string variable holding the URL for the image.

```
//Passing FileInputStream object as a parameter
FileInputStream inputstream = new FileInputStream("C:\\images\\image.jpg");
Image image = new Image(inputstream);

//Loading image from URL
/Image image = new Image(new FileInputStream("url for the image"));
```

After loading the image, you can set the view for the image by instantiating the **ImageView** class and passing the image to its constructor as follows –

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

Example

Following is an example which demonstrates how to load an image in JavaFX and set the view.

Save this code in a file with the name **ImageExample.java.**

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.image.Image;
```

```
import javafx.scene.image.ImageView;

import javafx.stage.Stage;

public class ImageExample extends Application {

    @Override
    public void start(Stage stage) throws FileNotFoundException {

        //Creating an image
        Image image = new Image(new FileInputStream("path of the image"));

        //Setting the image view
        ImageView imageView = new ImageView(image);
        //Setting the position of the image
        imageView.setX(50);
        imageView.setY(25);
        //setting the fit height and width of the image view
        imageView.setFitHeight(455);
        imageView.setFitWidth(500);
        //Setting the preserve ratio of the image view
        imageView.setPreserveRatio(true);

        //Creating a Group object
        Group root = new Group(imageView);

        //Creating a scene object
        Scene scene = new Scene(root, 600, 500);

        //Setting title to the Stage
        stage.setTitle("Loading an image");

        //Adding scene to the satge
        stage.setScene(scene);
```

```
        //Displaying the contents of the stage
        stage.show();
    }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
Javac ImageExample.java
java ImageExample
```

On executing, the above program generates a JavaFX window as follows –

Multiple Views of an Image

You can also set multiple views for an image in the same scene. The following program is an example that demonstrates how to set various views for an image in a scene in JavaFX.

Save this code in a file with the name **MultipleViews.java.**

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.image.Image;

import javafx.scene.image.ImageView;
import javafx.stage.Stage;

public class MultipleViews extends Application {

    @Override
    public void start(Stage stage) throws FileNotFoundException {

        //Creating an image
        Image image = new Image(new FileInputStream("file path"));

        //Setting the image view 1
        ImageView imageView1 = new ImageView(image);
        //Setting the position of the image
        imageView1.setX(50);
        imageView1.setY(25);
        //setting the fit height and width of the image view
        imageView1.setFitHeight(300);
        imageView1.setFitWidth(250);
        //Setting the preserve ratio of the image view
        imageView1.setPreserveRatio(true);

        //Setting the image view 2
        ImageView imageView2 = new ImageView(image);
```

```
        //Setting the position of the image
        imageView2.setX(350);
        imageView2.setY(25);
        //setting the fit height and width of the image view
        imageView2.setFitHeight(150);
        imageView2.setFitWidth(250);

        //Setting the preserve ratio of the image view
        imageView2.setPreserveRatio(true);

        //Setting the image view 3
        ImageView imageView3 = new ImageView(image);

        //Setting the position of the image
        imageView3.setX(350);
        imageView3.setY(200);
        //setting the fit height and width of the image view
        imageView3.setFitHeight(100);
        imageView3.setFitWidth(100);
        //Setting the preserve ratio of the image view
        imageView3.setPreserveRatio(true);

        //Creating a Group object
        Group root = new Group(imageView1, imageView2, imageView3);

        //Creating a scene object
        Scene scene = new Scene(root, 600, 400);

        //Setting title to the Stage
        stage.setTitle("Multiple views of an image");

        //Adding scene to the satge
        stage.setScene(scene);

        //Displaying the contents of the stage
        stage.show();
```

```
    }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
Javac MultipleViews.java
java MultipleViews
```

On executing, the above program generates a JavaFX window as follows –

Writing Pixels

JavaFX provides classes named **PixelReader** and **PixelWriter** classes to read and write pixels of an image. The **WritableImage** class is used to create a writable image.

Following is an example which demonstrates how to read and write pixels of an image. Here, we are reading the color value of an image and making it darker.

Save this code in a file with the name **WritingPixelsExample.java.**

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;

import javafx.application.Application;
import javafx.scene.Group;

import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.image.PixelReader;
import javafx.scene.image.PixelWriter;
import javafx.scene.image.WritableImage;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class WritingPixelsExample extends Application {

    @Override
    public void start(Stage stage) throws FileNotFoundException {

        //Creating an image
        Image image = new Image(new FileInputStream("C:\\images\\logo.jpg"));
        int width = (int)image.getWidth();
        int height = (int)image.getHeight();

        //Creating a writable image
        WritableImage wImage = new WritableImage(width, height);

        //Reading color from the loaded image
        PixelReader pixelReader = image.getPixelReader();
        //getting the pixel writer
        PixelWriter writer = wImage.getPixelWriter();
```

```
        //Reading the color of the image
        for(int y = 0; y < height; y++) {
           for(int x = 0; x < width; x++) {

             //Retrieving the color of the pixel of the loaded image
             Color color = pixelReader.getColor(x, y);

             //Setting the color to the writable image
             writer.setColor(x, y, color.darker());

             //Setting the view for the writable image
             ImageView imageView = new ImageView(wImage);

             //Creating a Group object
             Group root = new Group(imageView);

             //Creating a scene object
             Scene scene = new Scene(root, 600, 500);

             //Setting title to the Stage
             stage.setTitle("Writing pixels ");

             //Adding scene to the satge
             stage.setScene(scene);

             //Displaying the contents of the stage
             stage.show();

            }

        }
    }

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

```
}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
Javac WritingPixelsExample.java
java WritingPixelsExample
```

On executing, the above program generates a JavaFX window as follows –

12. JavaFX – 3D Shapes

In the earlier chapters, we have seen how to draw 2D shapes on an XY plane. In addition to these 2D shapes, we can draw several other 3D shapes as well using JavaFX.

3D Shape

In general, a 3D shape is a geometrical figure that can be drawn on the XYZ plane. These include a **Cylinder**, **Sphere** and a **Box**.

Each of the above mentioned 3D shape is represented by a class and all these classes belong to the package **javafx.scene.shape.** The class named **Shape3D** is the base class of all the 3-Dimensional shapes in JavaFX.

Creating a 3D Shape

To create a 3-Dimensional shape, you need to –

- Instantiate the respective class of the required 3D shape.
- Set the properties of the 3D shape.
- Add the 3D shape object to the group.

Instantiating the Respective Class

To create a 3-Dimensional shape, first of all you need to instantiate its respective class. For example, if you want to create a 3D box, you need to instantiate the class named **Box** as follows –

```
Box box = new Box();
```

Setting the Properties of the Shape

After instantiating the class, you need to set the properties for the shape using the setter methods.

For example, to draw a 3D box you need to pass its Width, Height, Depth. You can specify these values using their respective setter methods as follows —

```
//Setting the properties of the Box
box.setWidth(200.0);
box.setHeight(400.0);
box.setDepth(200.0);
```

Adding the Shape Object to the Group

Finally, you need to add the object of the shape to the group by passing it as a parameter of the constructor as shown below.

```
//Creating a Group object
Group root = new Group(box);
```

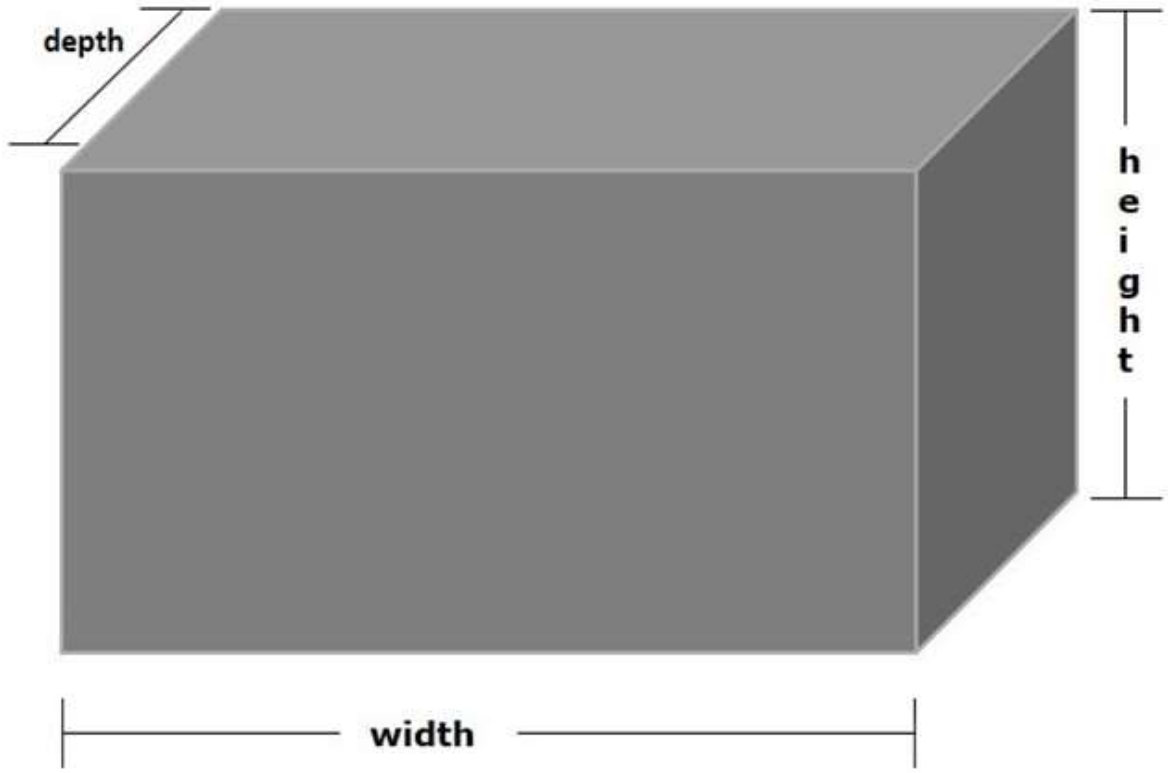
The following table gives you the list of various 3D shapes provided by JavaFX.

Shape	Description
Box	A cuboid is a three-dimensional shape with a **length** (depth), **width**, and a **height.** In JavaFX a three-dimensional box is represented by a class named **Box.** This class belongs to the package **javafx.scene.shape.** By instantiating this class, you can create a Box node in JavaFX. This class has 3 properties of the double datatype namely – • **width:** The width of the box. • **height:** The height of the box. • **depth:** The depth of the box.
Cylinder	A cylinder is a closed solid that has two parallel (mostly circular) bases connected by a curved surface. It is described by two parameters, namely, the **radius** of its circular base and the **height** of the cylinder. In JavaFX, a cylinder is represented by a class named **Cylinder.** This class belongs to the package **javafx.scene.shape**. By instantiating this class, you can create a cylinder node in JavaFX. This class has 2 properties of the double datatype namely – • **height:** The height of the Cylinder. • **radius:** The radius of the Cylinder.
Sphere	A sphere is defined as the set of points that are all at the same distance **r** from a given point in a 3D space. This distance **r** is the **radius** of the sphere and the given point is the centre of the sphere. In JavaFX, a sphere is represented by a class named **Sphere.** This class belongs to the package **javafx.scene.shape**. By instantiating this class, you can create a sphere node in JavaFX. This class has a property named **radius** of double datatype. It represents the radius of a Sphere.

3D Shapes – Box

A cuboid is a three dimensional or solid shape. Cuboids are made from 6 rectangles, which are placed at right angles. A cuboid that uses square faces is a cube, if the faces are rectangles, other than cubes, it looks like a shoe box.

A cuboid is a three-dimensional shape with a **length** (depth), **width**, and a **height** as shown in the following diagram —



In JavaFX, a 3-dimensional box is represented by a class named **Box.** This class belongs to the package **javafx.scene.shape**.

By instantiating this class, you can create a Box node in JavaFX.

This class has 3 properties of the double datatype, which are –

- **width:** The width of the box.
- **height:** The height of the box.
- **depth:** The depth of the box.

To draw a cubic curve, you need to pass values to these properties by passing them to the constructor of this class. This has to be done in the same order at the time of instantiation as shown below –

```
Box box = new Box(width, height, depth);
```

Or, by using their respective setter methods as follows –

```
setWidth(value);
```

```
setHeight(value);
setDepth(value);
```

To Draw a 3D box in JavaFX, follow the steps given below.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application** and implement the **start()** method of this class as follows:

```
public class ClassName extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {

    }
}
```

Step 2: Creating a Box

You can create a Box in JavaFX by instantiating the class named **BOX,** which belongs to a package **javafx.scene.shape**. You can instantiate this class as follows.

```
//Creating an object of the class Box
Box box = new Box();
```

Step 3: Setting Properties to the Box

Set the properties of the 3D box, **Width**, **Height** and **Depth**, using their respective setter methods as shown in the following code block.

```
//Setting the properties of the Box
box.setWidth(200.0);
box.setHeight(400.0);
box.setDepth(200.0);
```

Step 4: Creating a Group Object

In the **start()** method, create a group object by instantiating the class named **Group**, which belongs to the package **javafx.scene**.

Pass the Box (node) object, created in the previous step, as a parameter to the constructor of the Group class. This should be done in order to add it to the group as follows –

```
Group root = new Group(box);
```

Step 5: Creating a Scene Object

Create a Scene by instantiating the class named **Scene**, which belongs to the package **javafx.scene.** To this class, pass the Group object (**root**), created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen along with the object of the Group class as follows:

```
Scene scene = new Scene(group ,600, 300);
```

Step 6: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object, which is passed to the start method of the scene class as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 7: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using the following method:

```
primaryStage.setScene(scene);
```

Step 8: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 9: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows:

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

Example

Following is a program which generates a 3D box using JavaFX. Save this code in a file with the name **BoxExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.shape.Box;
import javafx.stage.Stage;

public class BoxExample extends Application {
   @Override
   public void start(Stage stage) {

      //Drawing a Box
      Box box = new Box();

      //Setting the properties of the Box
      box.setWidth(200.0);
      box.setHeight(400.0);
      box.setDepth(200.0);

      //Creating a Group object
      Group root = new Group(box);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);



      //Setting title to the Stage
      stage.setTitle("Drawing a Box");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }
```

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

Compile and execute the saved java file from the command prompt using the following commands.

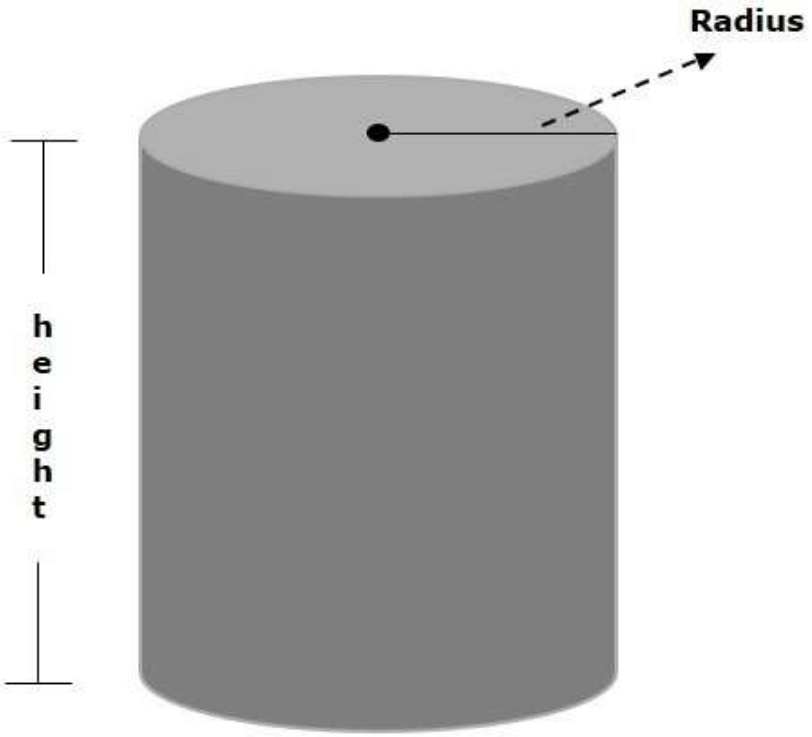
```
javac BoxExample.java
java BoxExample
```

On executing, the above program generates a JavaFX window displaying a 3D Box as shown below —

3D Shapes – Cylinder

A cylinder is a closed solid that has two parallel (mostly circular) bases connected by a curved surface.

It is described by two parameters, namely – the **radius** of its circular base and the **height** of the cylinder as shown in the following diagram —



In JavaFX, a cylinder is represented by a class named **Cylinder.** This class belongs to the package **javafx.scene.shape.** By instantiating this class, you can create a cylinder node in JavaFX.

This class has 2 properties of the double datatype namely –

- **height:** The height of the Cylinder.
- **radius:** The radius of the Cylinder.

To draw a cylinder, you need to pass values to these properties by passing them to the constructor of this class. This can be done in the same order at the time of instantiation, as shown in the following program –

```
Cylinder cylinder = new Cylinder(radius, height);
```

Or, by using their respective setter methods as follows –

```
setRadius(value);
setHeight(value);
```

To Draw a Cylinder (3D) in JavaFX, follow the steps given below.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application** and implement the **start()** method of this class as follows:

```
public class ClassName extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
    }
}
```

Step 2: Creating a Cylinder

You can create a Cylinder in JavaFX by instantiating the class named Cylinder, which belongs to a package **javafx.scene.shape**. You can instantiate this class as follows:

```
//Creating an object of the Cylinder class
Cylinder cylinder = new Cylinder();
```

Step 3: Setting Properties to the Cylinder

Set the **height** and **radius** of the Cylinder using their respective setter as shown below.

```
//Setting the properties of the Cylinder
cylinder.setHeight(300.0f);
cylinder.setRadius(100.0f);
```

Step 4: Creating a Group Object

In the **start()** method, create a group object by instantiating the class named **Group**, which belongs to the package **javafx.scene**.

Pass the Cylinder (node) object created in the previous step as a parameter to the constructor of the Group class. This should be done in order to add it to the group as follows –

```
Group root = new Group(cylinder);
```

Step 5: Creating a Scene Object

Create a Scene by instantiating the class named **Scene**, which belongs to the package **javafx.scene.** To this class, pass the Group object (**root**) created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 6: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. This **primaryStage** is a Stage object, which is passed to the start method of the scene class as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 7: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using this method as follows.

```
primaryStage.setScene(scene);
```

Step 8: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 9: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows.

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

Example

The following program shows how to generate a Cylinder using JavaFX. Save this code in a file with the name **CylinderExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.shape.CullFace;
import javafx.scene.shape.Cylinder;
import javafx.stage.Stage;
```

```
public class CylinderExample extends Application {
   @Override
   public void start(Stage stage) {

      //Drawing a Cylinder
      Cylinder cylinder = new Cylinder();

      //Setting the properties of the Cylinder
      cylinder.setHeight(300.0f);
      cylinder.setRadius(100.0f);



      //Creating a Group object
      Group root = new Group(cylinder);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Drawing a cylinder");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }
    public static void main(String args[]){
       launch(args);
      }
}
```

Compile and execute the saved java file from the command prompt using the following commands.

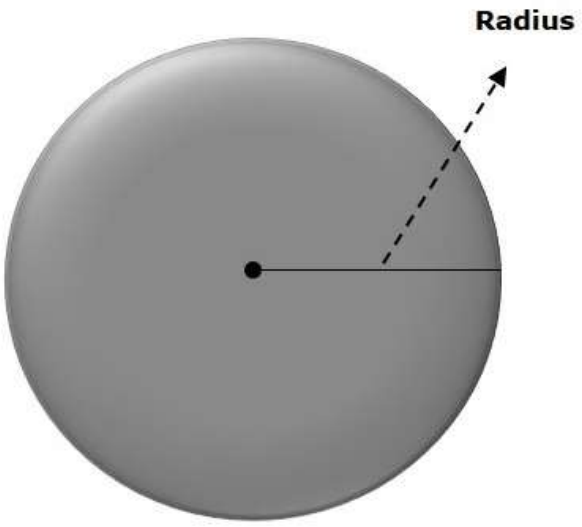
```
javac CylinderExample.java
java CylinderExample
```

On executing, the above program generates a JavaFX window displaying a Cylinder as shown below.

3D Shapes – Sphere

A sphere is a perfectly round geometrical object in a three-dimensional space that is the surface of a completely round shaped ball.

A sphere is defined as the set of points that are all at the same distance **r** from a given point in a 3D space. This distance **r** is the **radius** of the sphere and the given point is the centre of the sphere.



In JavaFX, a sphere is represented by a class named **Sphere.** This class belongs to the package **javafx.scene.shape.** By instantiating this class, you can create a sphere node in JavaFX.

This class has a property named **radius** of double datatype. It represents the radius of a Sphere. To draw a Sphere, you need to set values to this property by passing it to the constructor of this class at the time of instantiation as follows –

```
Sphere sphere = new Sphere(radius);
```

Or, by using a method named **setRadius()** as follows –

```
setRadius(value);
```

Follow the steps given below to Draw a Sphere (3D) in JavaFX.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application** and implement the **start()** method of this class as follows.

```
public class ClassName extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
    }
}
```

Step 2: Creating a Sphere

You can create a Sphere in JavaFX by instantiating the class named **Sphere**, which belongs to a package **javafx.scene.shape**. You can instantiate this class as follows.

```
//Creating an object of the class Sphere
Sphere sphere = new Sphere();
```

Step 3: Setting Properties to the Sphere

Set the radius of the Sphere using the method named **setRadius()** as shown below.

```
//Setting the radius of the Sphere
sphere.setRadius(300.0);
```

Step 4: Creating a Group Object

In the **start()** method, create a group object by instantiating the class named **Group**, which belongs to the package **javafx.scene**.

Pass the Sphere (node) object, created in the previous step, as a parameter to the constructor of the Group class. This should be done in order to add it to the group as shown below –

```
Group root = new Group(sphere);
```

Step 5: Creating a Scene Object

Create a Scene by instantiating the class named **Scene**, which belongs to the package **javafx.scene.** To this class, pass the Group object (**root**) created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 6: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object, which is passed to the start method of the scene class as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 7: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using this method as shown below.

```
primaryStage.setScene(scene);
```

Step 8: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 9: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as shown below.

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

Example

The following program shows how to generate a Sphere using JavaFX. Save this code in a file with the name **SphereExample.java.**

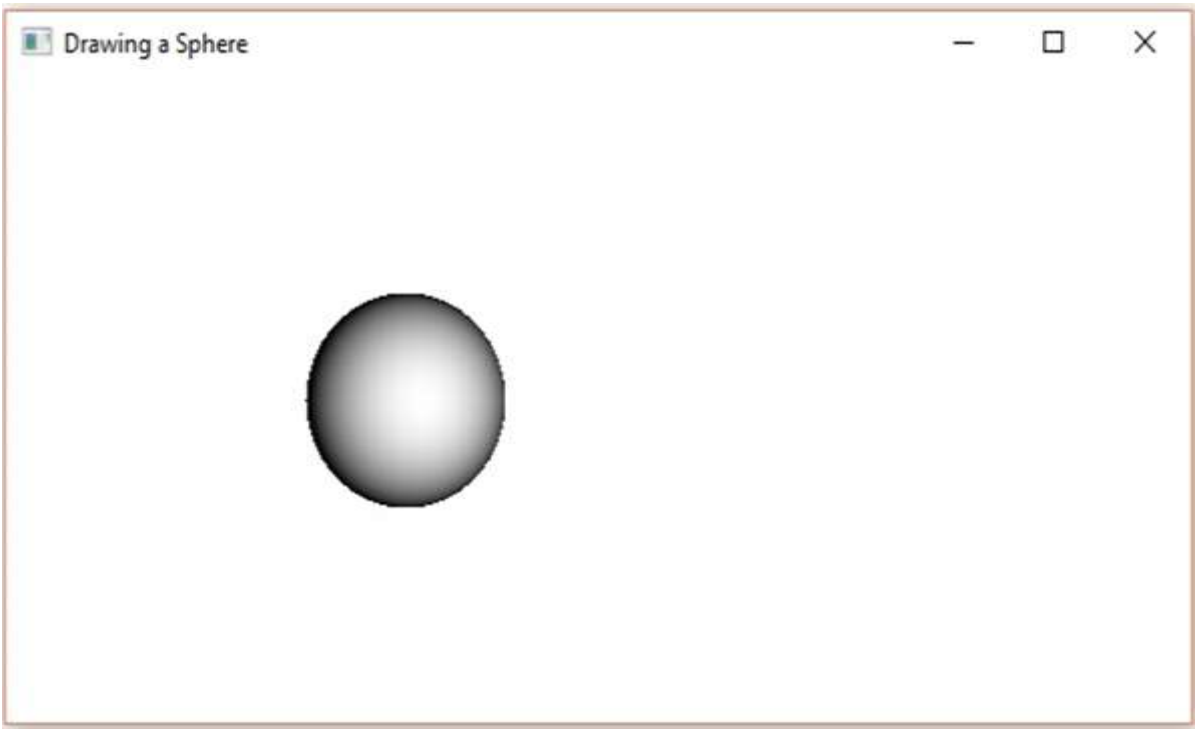
```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.shape.Sphere;

public class SphereExample extends Application {
   @Override
   public void start(Stage stage) {

      //Drawing a Sphere

      Sphere sphere = new Sphere();

      //Setting the properties of the Sphere
      sphere.setRadius(50.0);

      sphere.setTranslateX(200);
      sphere.setTranslateY(150);

      //Creating a Group object
      Group root = new Group(sphere);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting title to the Stage
      stage.setTitle("Drawing a Sphere - draw fill");
```

```
        //Adding scene to the satge
        stage.setScene(scene);

        //Displaying the contents of the stage
        stage.show();

    }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac SphereExample.java
java SphereExample
```

On executing, the above program generates a JavaFX window displaying a Sphere as shown below.



Properties of 3D Objects

For all the 3 Dimensional objects, you can set various properties like Cull Face, Drawing Mode, Material.

The following section discusses the properties of 3D objects.

Cull Face

In general, culling is the removal of improperly oriented parts of a shape (which are not visible in the view area).

The Cull Face property is of the type **CullFace** and it represents the Cull Face of a 3D shape. You can set the Cull Face of a shape using the method **setCullFace()** as shown below –

```
box.setCullFace(CullFace.NONE);
```

The stroke type of a shape can be –

- **None:** No culling is performed (CullFace.NONE).
- **Front:** All the front facing polygons are culled. (CullFace.FRONT).
- **Back:** All the back facing polygons are culled. (StrokeType.BACK).

By default, the cull face of a 3-Dimensional shape is Back.

Example

The following program is an example which demonstrates various cull faces of the sphere. Save this code in a file with the name **SphereCullFace.java**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.shape.CullFace;
import javafx.stage.Stage;
import javafx.scene.shape.Sphere;

public class SphereCullFace extends Application {
   @Override
   public void start(Stage stage) {

      //Drawing Sphere1
      Sphere sphere1 = new Sphere();
      //Setting the radius of the Sphere
```

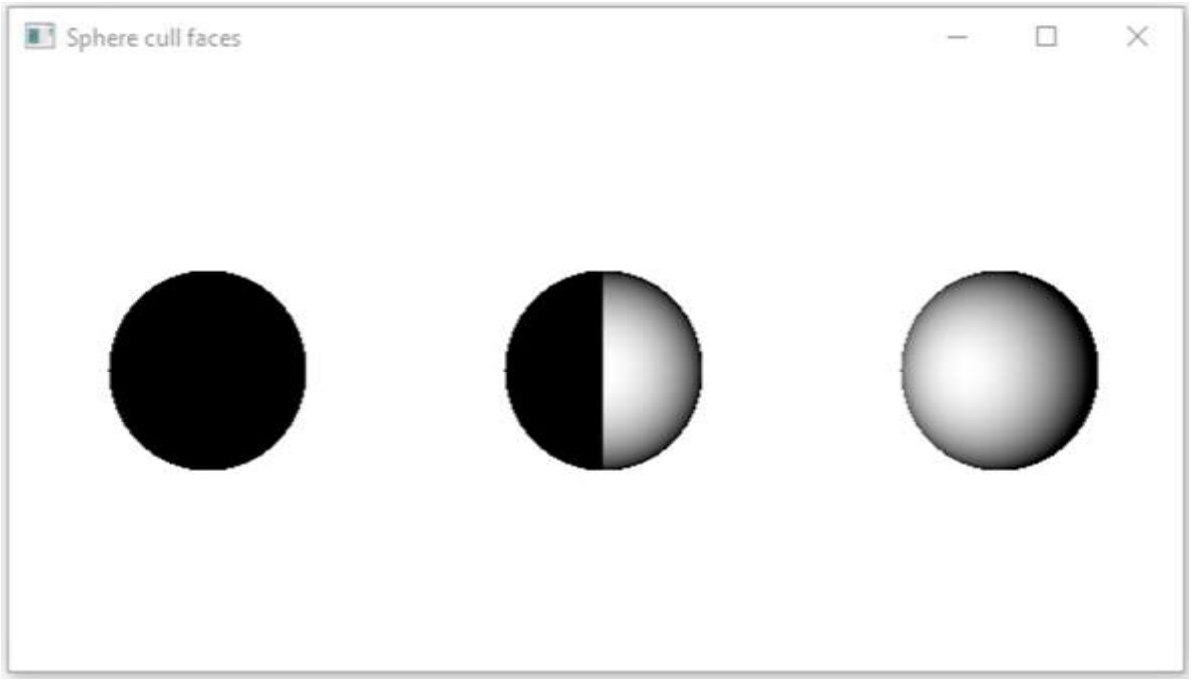
```
        sphere1.setRadius(50.0);
        //Settong the position of the sphere
        sphere1.setTranslateX(100);
        sphere1.setTranslateY(150);
        //setting the cull face of the sphere to front
        sphere1.setCullFace(CullFace.FRONT);

        //Drawing Sphere2
        Sphere sphere2 = new Sphere();
        //Setting the radius of the Sphere
        sphere2.setRadius(50.0);
        //Settong the position of the sphere
        sphere2.setTranslateX(300);

        sphere2.setTranslateY(150);
        //Setting the cull face of the sphere to back
        sphere2.setCullFace(CullFace.BACK);

        //Drawing Sphere3
        Sphere sphere3 = new Sphere();
        //Setting the radius of the Sphere
        sphere3.setRadius(50.0);
        //Settong the position of the sphere
        sphere3.setTranslateX(500);
        sphere3.setTranslateY(150);
        //Setting the cull face of the sphere to none
        sphere2.setCullFace(CullFace.NONE);

        //Creating a Group object
        Group root = new Group(sphere1, sphere2, sphere3);

        //Creating a scene object
        Scene scene = new Scene(root, 600, 300);

        //Setting title to the Stage
        stage.setTitle("Drawing a Sphere");
```

```
        //Adding scene to the satge
        stage.setScene(scene);

        //Displaying the contents of the stage
        stage.show();

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

Compile and execute the saved Java file from the command prompt using the following commands.

```
javac SphereCullFace.java
java SphereCullFace
```

On executing, the above program generates a JavaFX window displaying three spheres with cull face values **FRONT**, **BACK** and **NONE** respectively as follows –



Drawing Modes

It is the property is of the type **DrawMode** and it represents the drawing mode used to draw the current 3D shape. You can choose the draw mode to draw a 3D shape using the method setDrawMode () as follows –

```
box.setDrawMode(DrawMode.FILL);
```

In JavaFX, you can choose two draw modes to draw a 3D shape, which are –

- **Fill:** This mode draws and fills a 2D shape (DrawMode.FILL).
- **Line:** This mode draws a 3D shape using lines (DrawMode.LINE).

By default, the drawing mode of a 3Dimensional shape is fill.

Example

The following program is an example which demonstrates various draw modes of a 3D box. Save this code in a file with the name **BoxDrawMode.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.PerspectiveCamera;
import javafx.scene.Scene;

import javafx.scene.shape.Box;
import javafx.scene.shape.DrawMode;
import javafx.stage.Stage;

public class BoxDrawMode extends Application {
   @Override
   public void start(Stage stage) {

      //Drawing a Box
      Box box1 = new Box();
      //Setting the properties of the Box
      box1.setWidth(100.0);
      box1.setHeight(100.0);
      box1.setDepth(100.0);
      //Setting the position of the box
      box1.setTranslateX(200);
      box1.setTranslateY(150);
      box1.setTranslateZ(0);
```

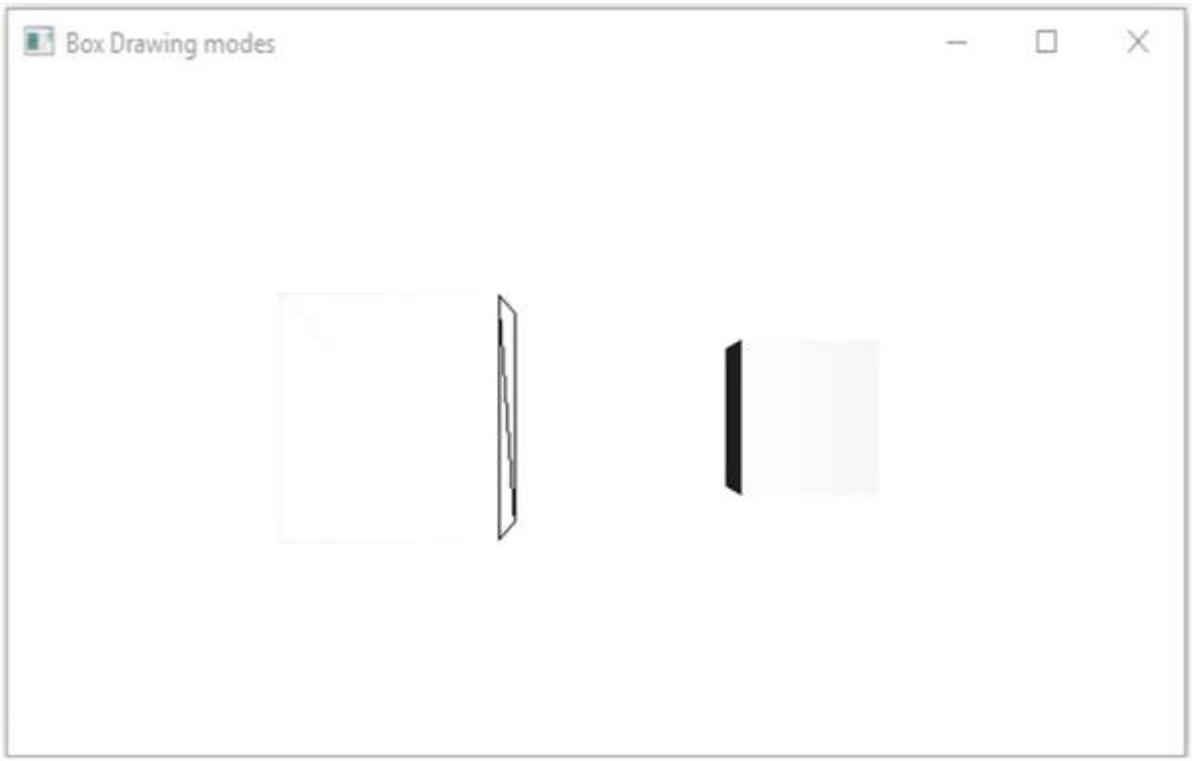
```
        //Setting the drawing mode of the box
        box1.setDrawMode(DrawMode.LINE);

        //Drawing a Box
        Box box2 = new Box();
        //Setting the properties of the Box
        box2.setWidth(100.0);
        box2.setHeight(100.0);
        box2.setDepth(100.0);
        //Setting the position of the box
        box2.setTranslateX(450); //450
        box2.setTranslateY(150);//150
        box2.setTranslateZ(300);

       //Setting the drawing mode of the box
        box2.setDrawMode(DrawMode.FILL);

        //Creating a Group object

        Group root = new Group(box1, box2);

        //Creating a scene object
        Scene scene = new Scene(root, 600, 300);

        //Setting camera
        PerspectiveCamera camera = new PerspectiveCamera(false);
        camera.setTranslateX(0);
        camera.setTranslateY(0);
        camera.setTranslateZ(0);
        scene.setCamera(camera);

        //Setting title to the Stage
        stage.setTitle("Drawing a Box");

        //Adding scene to the satge
        stage.setScene(scene);
```

```
      //Displaying the contents of the stage
      stage.show();

   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac BoxDrawMode.java
java BoxDrawMode
```

On executing, the above program generates a JavaFX window displaying two boxes with draw mode values LINE and FILL respectively, as follows –



Material

The cull Face property is of the type **Material** and it is used to choose the surface of the material of a 3D shape. You can set the material of a 3D shape using the method **setCullFace()** as follows –

```
cylinder.setMaterial(material);
```

As mentioned above for this method, you need to pass an object of the type Material. The **PhongMaterial** class of the package **javafx.scene.paint** is a sub class of this class and provides 7 properties that represent a Phong shaded material. You can apply all these type of materials to the surface of a 3D shape using the setter methods of these properties.

Following are the type of materials that are available in JavaFX –

- **bumpMap**: This represents a normal map stored as a RGB **Image**.
- **diffuseMap**: This represents a diffuse map.
- **selfIlluminationMap**: This represents a self-illumination map of this PhongMaterial.
- **specularMap**: This represents a specular map of this PhongMaterial.
- **diffuseColor**: This represents a diffuse color of this PhongMaterial.
- **specularColor**: This represents a specular color of this PhongMaterial.
- **specularPower**: This represents a specular power of this PhongMaterial.

By default, the material of a 3-Dimensional shape is a PhongMaterial with a diffuse color of light gray.

Example

Following is an example which displays various materials on the cylinder. Save this code in a file with the name **CylinderMaterials.java**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.PerspectiveCamera;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.paint.Color;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.shape.Cylinder;
import javafx.stage.Stage;

public class CylinderMaterials extends Application {
```

```
   @Override
   public void start(Stage stage) {

      //Drawing Cylinder1
      Cylinder cylinder1 = new Cylinder();

    //Setting the properties of the Cylinder
      cylinder1.setHeight(130.0f);
      cylinder1.setRadius(30.0f);

   //Setting the position of the Cylinder
      cylinder1.setTranslateX(100);
      cylinder1.setTranslateY(75);



      //Preparing the phong material of type bump map
      PhongMaterial material1 = new PhongMaterial();
      material1.setBumpMap(new Image("http://www.tutorialspoint.com/images/tp-
logo.gif"));
      //Setting the bump map material to Cylinder1
      cylinder1.setMaterial(material1);

      //Drawing Cylinder2
      Cylinder cylinder2 = new Cylinder();
      //Setting the properties of the Cylinder
      cylinder2.setHeight(130.0f);
      cylinder2.setRadius(30.0f);
      //Setting the position of the Cylinder
      cylinder2.setTranslateX(200);
      cylinder2.setTranslateY(75);

      //Preparing the phong material of type diffuse map
      PhongMaterial material2 = new PhongMaterial();
      material2.setDiffuseMap(new
Image("http://www.tutorialspoint.com/images/tp-logo.gif"));
      //Setting the diffuse map material to Cylinder2
```

```
        cylinder2.setMaterial(material2);

        //Drawing Cylinder3
        Cylinder cylinder3 = new Cylinder();
        //Setting the properties of the Cylinder
        cylinder3.setHeight(130.0f);
        cylinder3.setRadius(30.0f);

       //Setting the position of the Cylinder
        cylinder3.setTranslateX(300);
        cylinder3.setTranslateY(75);

        //Preparing the phong material of type Self Illumination Map
        PhongMaterial material3 = new PhongMaterial();
        material3.setSelfIlluminationMap(new
Image("http://www.tutorialspoint.com/images/tp-logo.gif"));

        //Setting the Self Illumination Map material to Cylinder3
        cylinder3.setMaterial(material3);

        //Drawing Cylinder4
        Cylinder cylinder4 = new Cylinder();
        //Setting the properties of the Cylinder
        cylinder4.setHeight(130.0f);
        cylinder4.setRadius(30.0f);
        //Setting the position of the Cylinder
        cylinder4.setTranslateX(400);
        cylinder4.setTranslateY(75);

        //Preparing the phong material of type Specular Map
        PhongMaterial material4 = new PhongMaterial();
        material4.setSpecularMap(new
Image("http://www.tutorialspoint.com/images/tp-logo.gif"));
        //Setting the Specular Map material to Cylinder4
        cylinder4.setMaterial(material4);

        //Drawing Cylinder5
```

```
Cylinder cylinder5 = new Cylinder();
//Setting the properties of the Cylinder
cylinder5.setHeight(130.0f);
cylinder5.setRadius(30.0f);
//Setting the position of the Cylinder
cylinder5.setTranslateX(100);
cylinder5.setTranslateY(300);

//Preparing the phong material of type diffuse color
PhongMaterial material5 = new PhongMaterial();
material5.setDiffuseColor(Color.BLANCHEDALMOND);
//Setting the diffuse color material to Cylinder5
cylinder5.setMaterial(material5);

//Drawing Cylinder6

Cylinder cylinder6 = new Cylinder();
//Setting the properties of the Cylinder
cylinder6.setHeight(130.0f);
cylinder6.setRadius(30.0f);
//Setting the position of the Cylinder
cylinder6.setTranslateX(200);
cylinder6.setTranslateY(300);

//Preparing the phong material of type specular color
PhongMaterial material6 = new PhongMaterial();
//setting the specular color map to the material
material6.setSpecularColor(Color.BLANCHEDALMOND);
//Setting the specular color material to Cylinder6
cylinder6.setMaterial(material6);

//Drawing Cylinder7
Cylinder cylinder7 = new Cylinder();
//Setting the properties of the Cylinder
cylinder7.setHeight(130.0f);
cylinder7.setRadius(30.0f);
```

```
      //Setting the position of the Cylinder
      cylinder7.setTranslateX(300);
      cylinder7.setTranslateY(300);

      //Preparing the phong material of type Specular Power
      PhongMaterial material7 = new PhongMaterial();
      material7.setSpecularPower(0.1);
      //Setting the Specular Power material to the Cylinder

     cylinder7.setMaterial(material7);

      //Creating a Group object
      Group root = new Group(cylinder1 ,cylinder2, cylinder3, cylinder4,
cylinder5, cylinder6, cylinder7);



      //Creating a scene object
      Scene scene = new Scene(root, 600, 400);

      //Setting camera
      PerspectiveCamera camera = new PerspectiveCamera(false);
      camera.setTranslateX(0);
      camera.setTranslateY(0);
      camera.setTranslateZ(-10);
      scene.setCamera(camera);

      //Setting title to the Stage
      stage.setTitle("Drawing a cylinder");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }
```

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

Compile and execute the saved java file from the command prompt using the following commands.

```
Javac CylinderMaterials.java
java CylinderMaterials
```

On executing, the above program generates a JavaFX window displaying 7 cylinders with Materials, Bump Map, Diffuse Map, Self-Illumination Map, Specular Map, Diffuse Color, Specular Color, (BLANCHEDALMOND) Specular Power, respectively, as shown in the following screenshot –

13. JavaFX – Event Handling

In JavaFX, we can develop GUI applications, web applications and graphical applications. In such applications, whenever a user interacts with the application (nodes), an event is said to have been occurred.

For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page are the activities that causes an event to happen.

Types of Events

The events can be broadly classified into the following two categories:

- **Foreground Events** – Those events which require the direct interaction of a user. They are generated as consequences of a person interacting with the graphical components in a Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page, etc.

- **Background Events** – Those events that require the interaction of end user are known as background events. The operating system interruptions, hardware or software failure, timer expiry, operation completion are the example of background events.

Events in JavaFX

JavaFX provides support to handle a wide varieties of events. The class named **Event** of the package **javafx.event** is the base class for an event.

An instance of any of its subclass is an event. JavaFX provides a wide variety of events. Some of them are are listed below.

- **Mouse Event:** This is an input event that occurs when a mouse is clicked. It is represented by the class named **MouseEvent**. It includes actions like mouse clicked, mouse pressed, mouse released, mouse moved, mouse entered target, mouse exited target, etc.

- **Key Event:** This is an input event that indicates the key stroke occurred on a node. It is represented by the class named **KeyEvent.** This event includes actions like key pressed, key released and key typed.

- **Drag Event:** This is an input event which occurs when the mouse is dragged. It is represented by the class named **DragEvent**. It includes actions like drag entered, drag dropped, drag entered target, drag exited target, drag over, etc.

- **Window Event:** This is an event related to window showing/hiding actions. It is represented by the class named **WindowEvent.** It includes actions like window hiding, window shown, window hidden, window showing, etc.

Event Handling

Event Handling is the mechanism that controls the event and decides what should happen, if an event occurs. This mechanism has the code which is known as an event handler that is executed when an event occurs.

JavaFX provides handlers and filters to handle events. In JavaFX every event has –

- **Target:** The node on which an event occurred. A target can be a window, scene, and a node.
- **Source:** The source from which the event is generated will be the source of the event. In the above scenario, mouse is the source of the event.
- **Type:** Type of the occurred event; in case of mouse event – mouse pressed, mouse released are the type of events.

Assume that we have an application which has a Circle, Stop and Play Buttons inserted using a group object as follows –

If you click on the play button, the source will be the mouse, the target node will be the play button and the type of the event generated is the mouse click.

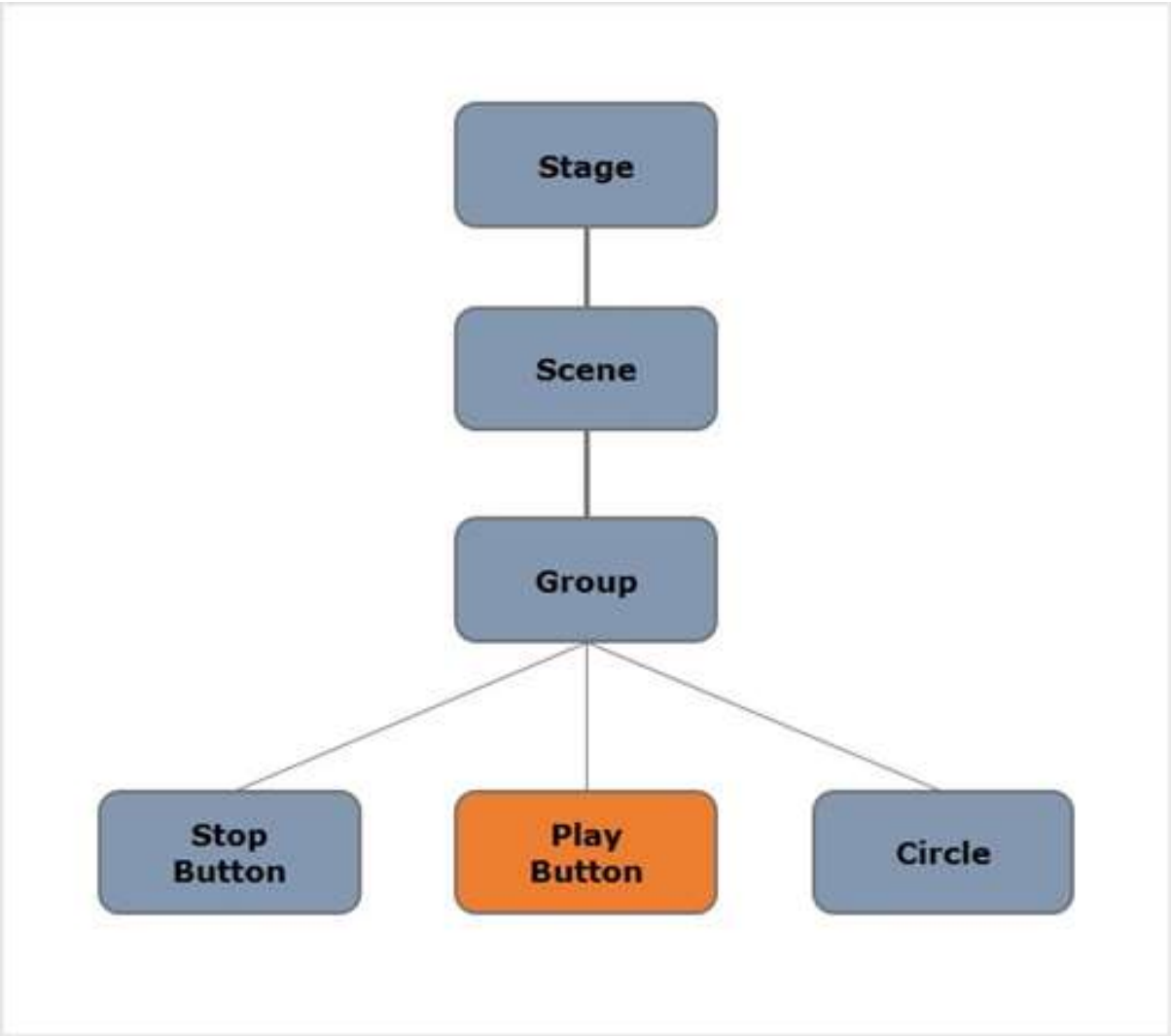
Phases of Event Handling in JavaFX

Whenever an event is generated, JavaFX undergoes the following phases.

Route Construction

Whenever an event is generated, the default/initial route of the event is determined by construction of an **Event Dispatch chain.** It is the path from the stage to the source Node.

Following is the event dispatch chain for the event generated, when we click on the play button in the above scenario.



Event Capturing Phase

After the construction of the event dispatch chain, the root node of the application dispatches the event. This event travels to all nodes in the dispatch chain (from top to bottom). If any of these nodes has a **filter** registered for the generated event, it will be executed. If none of the nodes in the dispatch chain has a filter for the event generated, then it is passed to the target node and finally the target node processes the event.

Event Bubbling Phase

In the event bubbling phase, the event is travelled from the target node to the stage node (bottom to top). If any of the nodes in the event dispatch chain has a **handler** registered for the generated event, it will be executed. If none of these nodes have handlers to handle the event, then the event reaches the root node and finally the process will be completed.

Event Handlers and Filters

Event filters and handlers are those which contains application logic to process an event. A node can register to more than one handler/filter. In case of parent–child nodes, you can provide a common filter/handler to the parents, which is processed as default for all the child nodes.

As mentioned above, during the event, processing is a filter that is executed and during the event bubbling phase, a handler is executed. All the handlers and filters implement the interface **EventHandler** of the package **javafx.event**.

Adding and Removing Event Filter

To add an event filter to a node, you need to register this filter using the method **addEventFilter()** of the **Node** class.

```
//Creating the mouse event handler
EventHandler<MouseEvent> eventHandler = new EventHandler<MouseEvent>() {
   @Override
   public void handle(MouseEvent e) {
      System.out.println("Hello World");
      circle.setFill(Color.DARKSLATEBLUE);
   }
};

//Adding event Filter
Circle.addEventFilter(MouseEvent.MOUSE_CLICKED, eventHandler);
```

In the same way, you can remove a filter using the method **removeEventFilter()** as shown below:

```
circle.removeEventFilter(MouseEvent.MOUSE_CLICKED, eventHandler);
```

Event Handling Example

Following is an example demonstrating the event handling in JavaFX using the event filters. Save this code in a file with name **EventFiltersExample.java**.

```
import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.event.EventHandler;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
```

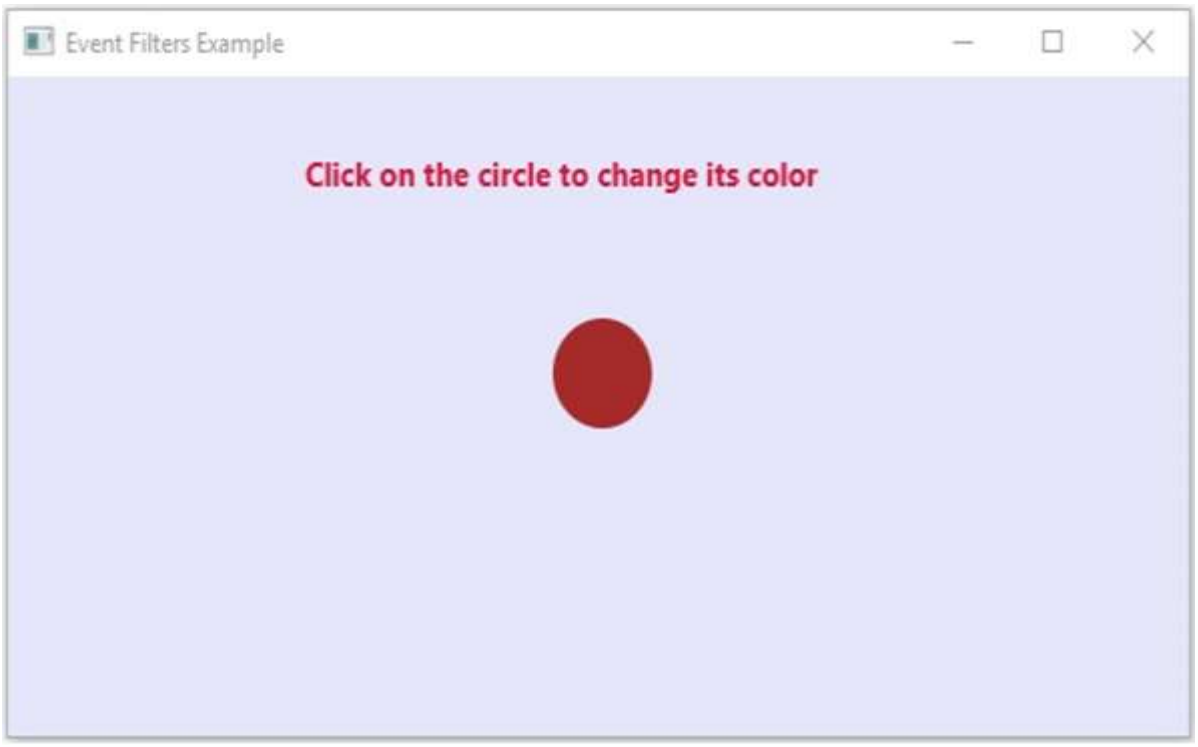
```
import javafx.stage.Stage;

public class EventFiltersExample extends Application {
   @Override
   public void start(Stage stage) {

      //Drawing a Circle
      Circle circle = new Circle();
      //Setting the position of the circle
      circle.setCenterX(300.0f);
      circle.setCenterY(135.0f);
      //Setting the radius of the circle
      circle.setRadius(25.0f);
      //Setting the color of the circle
      circle.setFill(Color.BROWN);
      //Setting the stroke width of the circle
      circle.setStrokeWidth(20);

      //Setting the text
      Text text = new Text("Click on the circle to change its color");
      //Setting the font of the text
      text.setFont(Font.font(null, FontWeight.BOLD, 15));
      //Setting the color of the text
      text.setFill(Color.CRIMSON);

     //setting the position of the text
      text.setX(150);
      text.setY(50);

      //Creating the mouse event handler
      EventHandler<MouseEvent> eventHandler = new EventHandler<MouseEvent>() {
         @Override
         public void handle(MouseEvent e) {
            System.out.println("Hello World");
            circle.setFill(Color.DARKSLATEBLUE);
```

```
         }
      };

      //Regestering the event filter
      circle.addEventFilter(MouseEvent.MOUSE_CLICKED, eventHandler);

      //Creating a Group object
      Group root = new Group(circle, text);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);

      //Setting the fill color to the scene
      scene.setFill(Color.LAVENDER);

      //Setting title to the Stage
      stage.setTitle("Event Filters Example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();


   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac EventFiltersExample.java
java EventFiltersExample
```

On executing, the above program generates a JavaFX window as shown below.



Adding and Removing Event Handlers

To add an event handler to a node, you need to register this handler using the method **addEventHandler()** of the **Node** class as shown below.

```
//Creating the mouse event handler
EventHandler<javafx.scene.input.MouseEvent> eventHandler = new
EventHandler<javafx.scene.input.MouseEvent>() {
   @Override
   public void handle(javafx.scene.input.MouseEvent e) {
      System.out.println("Hello World");
      circle.setFill(Color.DARKSLATEBLUE);
     }
};

//Adding the event handler
circle.addEventHandler(javafx.scene.input.MouseEvent.MOUSE_CLICKED,
eventHandler);
```

In the same way, you can remove an event handler using the method removeEventHandler() as shown below:

```
circle.removeEventHandler(MouseEvent.MOUSE_CLICKED, eventHandler);
```

Example

The following program is an example demonstrating the event handling in JavaFX using the event handlers.

Save this code in a file with name **EventHandlersExample.java**.

```
import javafx.animation.RotateTransition;
import javafx.application.Application;
import javafx.event.EventHandler;
import javafx.scene.Group;
import javafx.scene.PerspectiveCamera;
import javafx.scene.Scene;
import javafx.scene.control.TextField;
import javafx.scene.input.KeyEvent;
import javafx.scene.paint.Color;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.shape.Box;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;

import javafx.scene.transform.Rotate;
import javafx.stage.Stage;
import javafx.util.Duration;

public class EventHandlersExample extends Application {
   @Override
   public void start(Stage stage) {
      //Drawing a Box
      Box box = new Box();
      //Setting the properties of the Box
      box.setWidth(150.0);
      box.setHeight(150.0);
      box.setDepth(100.0);

      //Setting the position of the box
```

```
        box.setTranslateX(350);
        box.setTranslateY(150);
        box.setTranslateZ(50);

        //Setting the text
        Text text = new Text("Type any letter to rotate the box, and click on
the box to stop the rotation");
        //Setting the font of the text
        text.setFont(Font.font(null, FontWeight.BOLD, 15));
        //Setting the color of the text
        text.setFill(Color.CRIMSON);
        //setting the position of the text
        text.setX(20);
        text.setY(50);

        //Setting the material of the box
        PhongMaterial material = new PhongMaterial();
        material.setDiffuseColor(Color.DARKSLATEBLUE);

        //Setting the diffuse color material to box
        box.setMaterial(material);

        //Setting the rotation animation to the box
        RotateTransition rotateTransition = new RotateTransition();
        //Setting the duration for the transition
        rotateTransition.setDuration(Duration.millis(1000));
        //Setting the node for the transition
        rotateTransition.setNode(box);
        //Setting the axis of the rotation
        rotateTransition.setAxis(Rotate.Y_AXIS);
        //Setting the angle of the rotation
        rotateTransition.setByAngle(360);
        //Setting the cycle count for the transition
        rotateTransition.setCycleCount(50);
        //Setting auto reverse value to false
        rotateTransition.setAutoReverse(false);
```

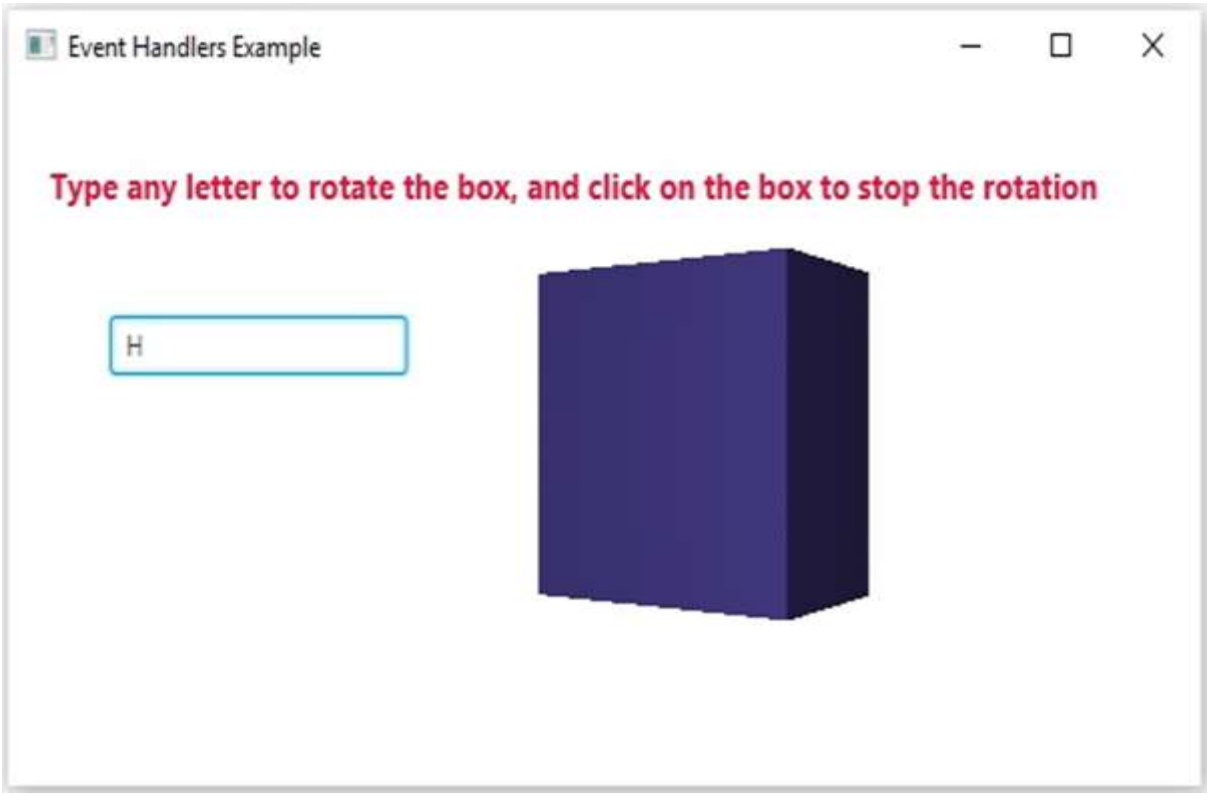
```
      //Creating a text filed
      TextField textField = new TextField();
      //Setting the position of the text field
      textField.setLayoutX(50);
      textField.setLayoutY(100);

      //Handling the key typed event
      EventHandler<KeyEvent> eventHandlerTextField = new
EventHandler<KeyEvent>() {
          @Override
          public void handle(KeyEvent event) {
             //Playing the animation
             rotateTransition.play();
          }
      };
      //Adding an event handler to the text feld
      textField.addEventHandler(KeyEvent.KEY_TYPED, eventHandlerTextField);

      //Handling the mouse clicked event(on box)
      EventHandler<javafx.scene.input.MouseEvent> eventHandlerBox = new
EventHandler<javafx.scene.input.MouseEvent>() {
         @Override
         public void handle(javafx.scene.input.MouseEvent e) {
            rotateTransition.stop();
         }
      };
      //Adding the event handler to the box
      box.addEventHandler(javafx.scene.input.MouseEvent.MOUSE_CLICKED,
eventHandlerBox);

      //Creating a Group object
      Group root = new Group(box, textField, text);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);
```

```
      //Setting camera
      PerspectiveCamera camera = new PerspectiveCamera(false);
      camera.setTranslateX(0);
      camera.setTranslateY(0);
      camera.setTranslateZ(0);
      scene.setCamera(camera);

      //Setting title to the Stage
      stage.setTitle("Event Handlers Example");

      //Adding scene to the stage
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac EventHandlersExample.java
java EventHandlersExample
```

On executing, the above program generates a JavaFX window displaying a text field and a 3D box as shown below –



Here, if you type a letter in the text field, the 3D box starts rotating along the x axis. If you click on the box again the rotation stops.

Using Convenience Methods for Event Handling

Some of the classes in JavaFX define event handler properties. By setting the values to these properties using their respective setter methods, you can register to an event handler. These methods are known as convenience methods.

Most of these methods exist in the classes like Node, Scene, Window, etc., and they are available to all their sub classes.

For example, to add a mouse event listener to a button, you can use the convenience method **setOnMouseClicked()** as shown below.

```
playButton.setOnMouseClicked((new EventHandler<MouseEvent>() {
   public void handle(MouseEvent event) {
      System.out.println("Hello World");
      pathTransition.play();
      }
}));
```

Example

The following program is an example that demonstrates the event handling in JavaFX using the convenience methods.

Save this code in a file with the name **ConvenienceMethodsExample.java**.

```
import javafx.animation.PathTransition;
import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.event.EventHandler;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.LineTo;
import javafx.scene.shape.MoveTo;
import javafx.scene.shape.Path;
import javafx.stage.Stage;
import javafx.util.Duration;

public class ConvenienceMethodsExample extends Application {
   @Override
   public void start(Stage stage) {

      //Drawing a Circle
      Circle circle = new Circle();
      //Setting the position of the circle
      circle.setCenterX(300.0f);
      circle.setCenterY(135.0f);
      //Setting the radius of the circle
      circle.setRadius(25.0f);

      //Setting the color of the circle
      circle.setFill(Color.BROWN);
      //Setting the stroke width of the circle
      circle.setStrokeWidth(20);
```

```
        //Creating a Path
        Path path = new Path();
        //Moving to the starting point
        MoveTo moveTo = new MoveTo(208, 71);
        //Creating 1st line
        LineTo line1 = new LineTo(421, 161);
        //Creating 2nd line
        LineTo line2 = new LineTo(226,232);
        //Creating 3rd line
        LineTo line3 = new LineTo(332,52);
        //Creating 4th line
        LineTo line4 = new LineTo(369, 250);
        //Creating 5th line
        LineTo line5 = new LineTo(208, 71);
        //Adding all the elements to the path
        path.getElements().add(moveTo);
        path.getElements().addAll(line1, line2, line3, line4, line5);


        //Creating the path transition
        PathTransition pathTransition = new PathTransition();
        //Setting the duration of the transition
        pathTransition.setDuration(Duration.millis(1000));
        //Setting the node for the transition
        pathTransition.setNode(circle);
        //Setting the path for the transition
        pathTransition.setPath(path);
        //Setting the orientation of the path


pathTransition.setOrientation(PathTransition.OrientationType.ORTHOGONAL_TO_TAN
GENT);
        //Setting the cycle count for the transition
        pathTransition.setCycleCount(50);
```

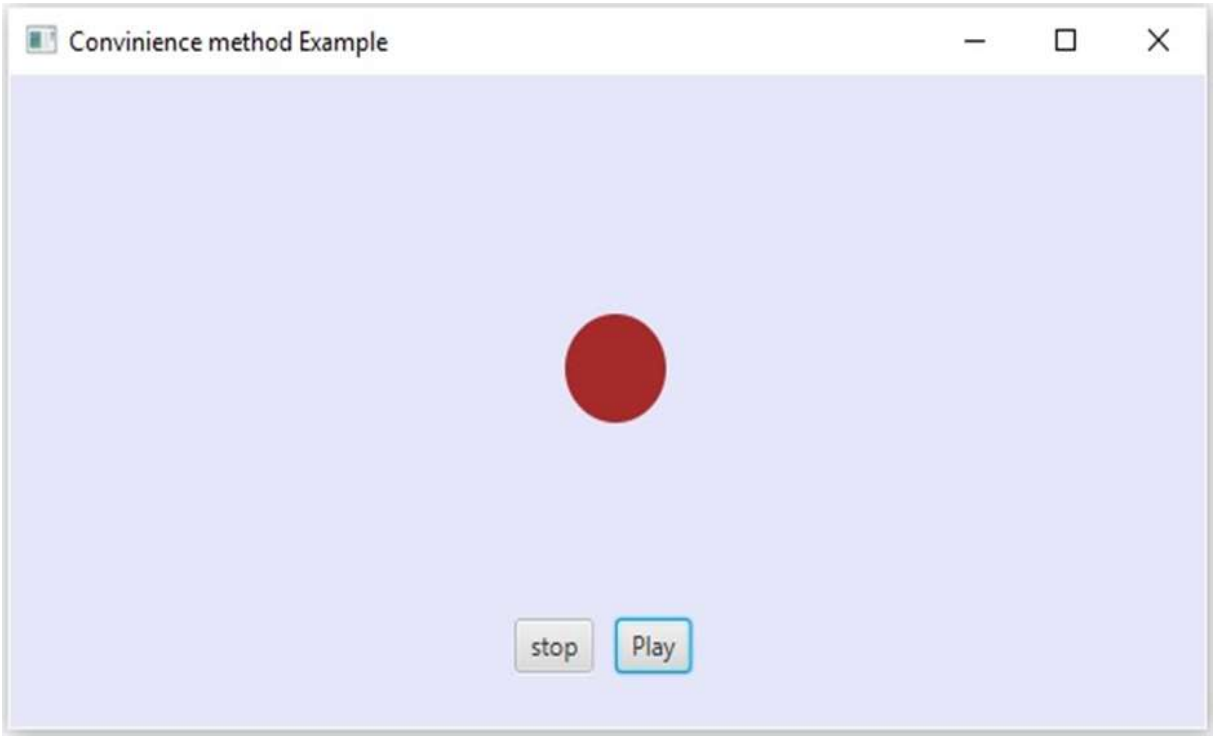
```
      //Setting auto reverse value to true
      pathTransition.setAutoReverse(false);

      //Creating play button
      Button playButton = new Button("Play");
      playButton.setLayoutX(300);
      playButton.setLayoutY(250);

      circle.setOnMouseClicked(new
EventHandler<javafx.scene.input.MouseEvent>() {
         @Override
         public void handle(javafx.scene.input.MouseEvent e) {
            System.out.println("Hello World");
            circle.setFill(Color.DARKSLATEBLUE);
         }
      });

      playButton.setOnMouseClicked((new EventHandler<MouseEvent>() {

            public void handle(MouseEvent event) {
                System.out.println("Hello World");

                pathTransition.play();
            }
        }));

      //Creating stop button
      Button stopButton = new Button("stop");
      stopButton.setLayoutX(250);
      stopButton.setLayoutY(250);

      stopButton.setOnMouseClicked((new EventHandler<MouseEvent>() {

            public void handle(MouseEvent event) {
                System.out.println("Hello World");
                pathTransition.stop();
```

```
            }
        }));

      //Creating a Group object
      Group root = new Group(circle, playButton, stopButton);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 300);
      scene.setFill(Color.LAVENDER);

      //Setting title to the Stage
      stage.setTitle("Convenience Methods Example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac ConvenienceMethodsExample.java
java ConvenienceMethodsExample
```

On executing, the above program generates a JavaFX window as shown below. Here click on the play button to start the animation and click on the stop button to stop the animation.



14. JavaFX – Controls (UI controls)

Every user interface considers the following three main aspects:

- **UI elements**: These are the core visual elements which the user eventually sees and interacts with. JavaFX provides a huge list of widely used and common elements varying from basic to complex, which we will cover in this tutorial.

- **Layouts:** They define how UI elements should be organized on the screen and provide a final look and feel to the GUI (Graphical User Interface). This part will be covered in the Layout chapter.

- **Behavior:** These are events which occur when the user interacts with UI elements. This part will be covered in the Event Handling chapter.

JavaFX provides several classes in the package **javafx.scene.control**. To create various GUI components (controls), JavaFX supports several controls such as date picker, button text field, etc.

Each control is represented by a class; you can create a control by instantiating its respective class.

Following is the list of commonly used controls while the GUI is designed using JavaFX.

Sr. No.	Control & Description
1	**Label** A Label object is a component for placing text.
2	**Button** This class creates a labeled button.
3	**ColorPicker** A ColorPicker provides a pane of controls designed to allow a user to manipulate and select a color.
4	**CheckBox** A CheckBox is a graphical component that can be in either an on(true) or off (false) state.

5	**RadioButton** The RadioButton class is a graphical component, which can either be in a ON (true) or OFF (false) state in a group.
6	**ListView** A ListView component presents the user with a scrolling list of text items.
7	**TextField** A TextField object is a text component that allows for the editing of a single line of text.
8	**PasswordField** A PasswordField object is a text component specialized for password entry.
9	**Scrollbar** A Scrollbar control represents a scroll bar component in order to enable user to select from range of values.
10	**FileChooser** A FileChooser control represents a dialog window from which the user can select a file.
11	**ProgressBar** As the task progresses towards completion, the progress bar displays the task's percentage of completion.
12	**Slider** A Slider lets the user graphically select a value by sliding a knob within a bounded interval.

Example

The following program is an example which displays a login page in JavaFX. Here, we are using the controls **label**, **text field**, **password field** and **button**.

Save this code in a file with the name **LoginPage.java**

```
import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.PasswordField;
import javafx.scene.layout.GridPane;
import javafx.scene.text.Text;
import javafx.scene.control.TextField;
import javafx.stage.Stage;


public class LoginPage extends Application {

   @Override
   public void start(Stage stage) {

      //creating label email
      Text text1 = new Text("Email");
      //creating label password
      Text text2 = new Text("Password");

      //Creating Text Filed for email
      TextField textField1 = new TextField();
      //Creating Text Filed for password
      PasswordField textField2 = new PasswordField();

      //Creating Buttons
      Button button1 = new Button("Submit");
      Button button2 = new Button("Clear");

      //Creating a Grid Pane
      GridPane gridPane = new GridPane();
      //Setting size for the pane
      gridPane.setMinSize(400, 200);
```

```
      //Setting the padding
      gridPane.setPadding(new Insets(10, 10, 10, 10));
      //Setting the vertical and horizontal gaps between the columns
      gridPane.setVgap(5);
      gridPane.setHgap(5);
      //Setting the Grid allignment
      gridPane.setAlignment(Pos.CENTER);

      //Arrianging all the nodes in the grid
      gridPane.add(text1, 0, 0);
      gridPane.add(textField1, 1, 0);
      gridPane.add(text2, 0, 1);
      gridPane.add(textField2, 1, 1);
      gridPane.add(button1, 0, 2);
      gridPane.add(button2, 1, 2);

      //Styling nodes
      button1.setStyle("-fx-background-color: darkslateblue; -fx-text-fill:
white;");
      button2.setStyle("-fx-background-color: darkslateblue; -fx-text-fill:
white;");

      text1.setStyle("-fx-font: normal bold 20px 'serif' ");
      text2.setStyle("-fx-font: normal bold 20px 'serif' ");

      gridPane.setStyle("-fx-background-color: BEIGE;");

      //Creating a scene object
      Scene scene = new Scene(gridPane);

      //Setting title to the Stage
      stage.setTitle("CSS Example");

      //Adding scene to the satge
      stage.setScene(scene);
```

```
      //Displaying the contents of the stage
      stage.show();

   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac LoginPage.java
java LoginPage
```

On executing, the above program generates a JavaFX window as shown below.

The following program is an example of a registration form, which demonstrates controls in JavaFX such as **Date Picker**, **Radio Button**, **Toggle Button**, **Check Box**, **List View**, **Choice List**, etc.

Save this code in a file with the name **Registration.java**.

```
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.CheckBox;
import javafx.scene.control.ChoiceBox;
import javafx.scene.control.DatePicker;
import javafx.scene.control.ListView;
import javafx.scene.control.RadioButton;
import javafx.scene.layout.GridPane;
import javafx.scene.text.Text;
import javafx.scene.control.TextField;
import javafx.scene.control.ToggleGroup;

import javafx.scene.control.ToggleButton;
import javafx.stage.Stage;

public class Registration extends Application {

   @Override
   public void start(Stage stage) {

      //Label for name
      Text nameLabel = new Text("Name");
      //Text field for name
      TextField nameText = new TextField();

      //Label for date of bith
      Text dobLabel = new Text("Date of birth");
      //date picker to chosse date
      DatePicker datePicker = new DatePicker();
```

```
        //Label for gender
        Text genderLabel = new Text("gender");
        //Toggle group of radio buttons
        ToggleGroup groupGender = new ToggleGroup();
        RadioButton maleRadio = new RadioButton("male");
        maleRadio.setToggleGroup(groupGender);
        RadioButton femaleRadio = new RadioButton("female");
        femaleRadio.setToggleGroup(groupGender);

        //Label for reservation
        Text reservationLabel = new Text("Reservation");
        //Toggle button for reservation
        ToggleButton Reservation = new ToggleButton();
        ToggleButton yes = new ToggleButton("Yes");
        ToggleButton no = new ToggleButton("No");
        ToggleGroup groupReservation = new ToggleGroup();
        yes.setToggleGroup(groupReservation);



        no.setToggleGroup(groupReservation);

        //Label for technologies known
        Text technologiesLabel = new Text("Technologies Known");
        //check box for education
        CheckBox javaCheckBox = new CheckBox("Java");
        javaCheckBox.setIndeterminate(false);
        //check box for education
        CheckBox dotnetCheckBox = new CheckBox("DotNet");
        javaCheckBox.setIndeterminate(false);

        //Label for education
        Text educationLabel = new Text("Educational qualification");
        //list View for educational qualification
        ObservableList<String> names = FXCollections.observableArrayList(
            "Engineering", "MCA", "MBA", "Graduation", "MTECH", "Mphil", "Phd");
        ListView<String> educationListView = new ListView<String>(names);
```

```
      //Label for location
      Text locationLabel = new Text("location");
      //Choice box for location
      ChoiceBox locationchoiceBox= new ChoiceBox();
      locationchoiceBox.getItems().addAll("Hyderabad", "Chennai", "Delhi",
"Mumbai", "Vishakhapatnam");

      //Label for register
      Button buttonRegister = new Button("Register");

      //Creating a Grid Pane
      GridPane gridPane = new GridPane();
      //Setting size for the pane
      gridPane.setMinSize(500, 500);

      //Setting the padding


      gridPane.setPadding(new Insets(10, 10, 10, 10));

      //Setting the vertical and horizontal gaps between the columns
      gridPane.setVgap(5);
      gridPane.setHgap(5);
      //Setting the Grid allignment
      gridPane.setAlignment(Pos.CENTER);

      //Arrianging all the nodes in the grid
      gridPane.add(nameLabel, 0, 0);
      gridPane.add(nameText, 1, 0);

      gridPane.add(dobLabel, 0, 1);
      gridPane.add(datePicker, 1, 1);

      gridPane.add(genderLabel, 0, 2);
      gridPane.add(maleRadio, 1, 2);
```

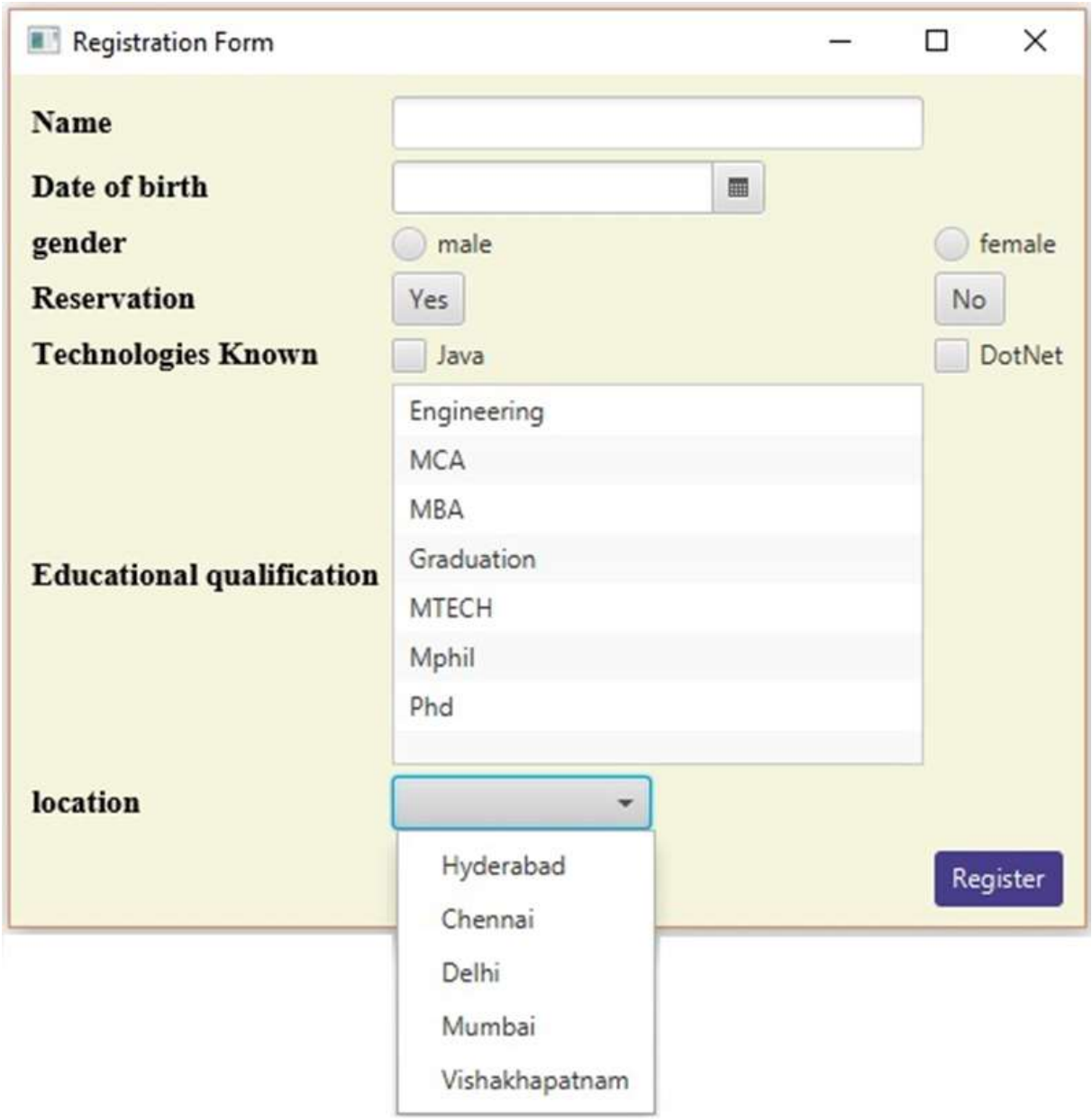
```
      gridPane.add(femaleRadio, 2, 2);

      gridPane.add(reservationLabel, 0, 3);
      gridPane.add(yes, 1, 3);
      gridPane.add(no, 2, 3);

      gridPane.add(technologiesLabel, 0, 4);
      gridPane.add(javaCheckBox, 1, 4);
      gridPane.add(dotnetCheckBox, 2, 4);

      gridPane.add(educationLabel, 0, 5);
      gridPane.add(educationListView, 1, 5);

      gridPane.add(locationLabel, 0, 6);
      gridPane.add(locationchoiceBox, 1, 6);

      gridPane.add(buttonRegister, 2, 8);

      //Styling nodes

      buttonRegister.setStyle("-fx-background-color: darkslateblue; -fx-text-
fill: white;");

      nameLabel.setStyle("-fx-font: normal bold 15px 'serif' ");
      dobLabel.setStyle("-fx-font: normal bold 15px 'serif' ");
      genderLabel.setStyle("-fx-font: normal bold 15px 'serif' ");
      reservationLabel.setStyle("-fx-font: normal bold 15px 'serif' ");
      technologiesLabel.setStyle("-fx-font: normal bold 15px 'serif' ");
      educationLabel.setStyle("-fx-font: normal bold 15px 'serif' ");
      locationLabel.setStyle("-fx-font: normal bold 15px 'serif' ");

      //Setting the back ground color
      gridPane.setStyle("-fx-background-color: BEIGE;");

      //Creating a scene object
      Scene scene = new Scene(gridPane);
```

```
      //Setting title to the Stage
      stage.setTitle("Registration Form");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac Registration.java
java Registration
```

On executing, the above program generates a JavaFX window as shown below.

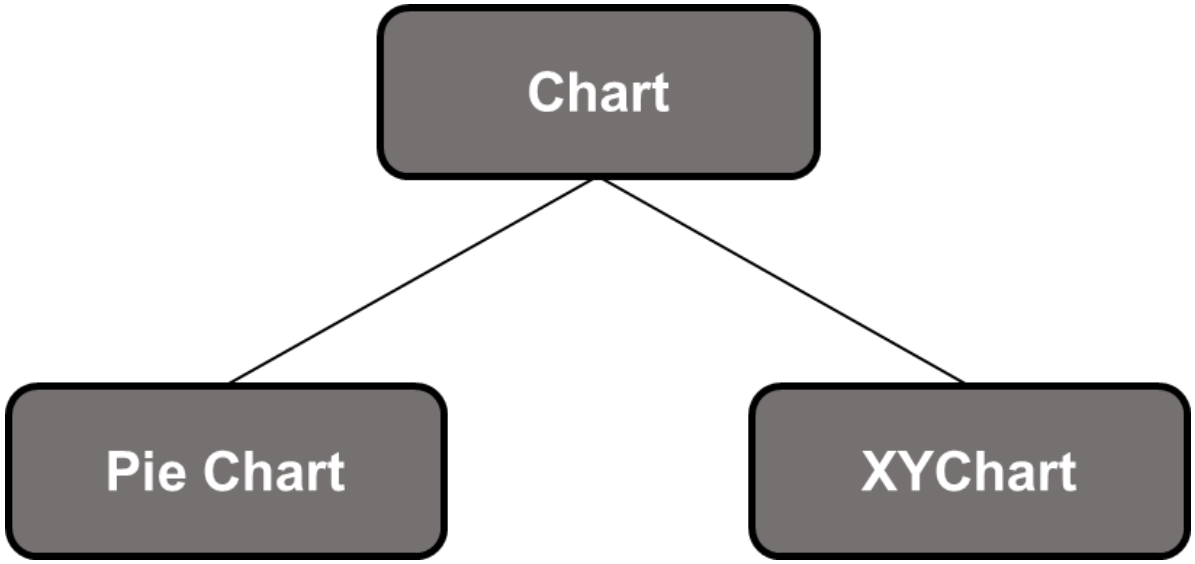


15. JavaFX – Charts

In general, a chart is a graphical representation of data. There are various kinds of charts to represent data such as **Bar Chart**, **Pie Chart**, **Line Chart**, **Scatter Chart**, etc.

JavaFX Provides support for various **Pie Charts** and **XY Charts**. The charts that are represented on an XY–plane include **AreaChart**, **BarChart**, **BubbleChart**, **LineChart**, **ScatterChart**, **StackedAreaChart**, **StackedBarChart**, etc.

Each chart is represented by a class and all these charts belongs to the package **javafx.scene.chart**. The class named **Chart** is the base class of all the charts in JavaFX and the **XYChart** is base class of all those charts that are drawn on the XY–plane.



Creating a Chart

To create a chart, you need to –

- Define the axis of the chart
- Instantiate the respective class
- Prepare and pass data to the chart

Instantiating the Respective Class

To create a chart, instantiate its respective class. For example, if you want to create a line chart, you need to instantiate the class named **Line** as follows –

```
LineChart linechart = new LineChart(xAxis, yAxis);
```

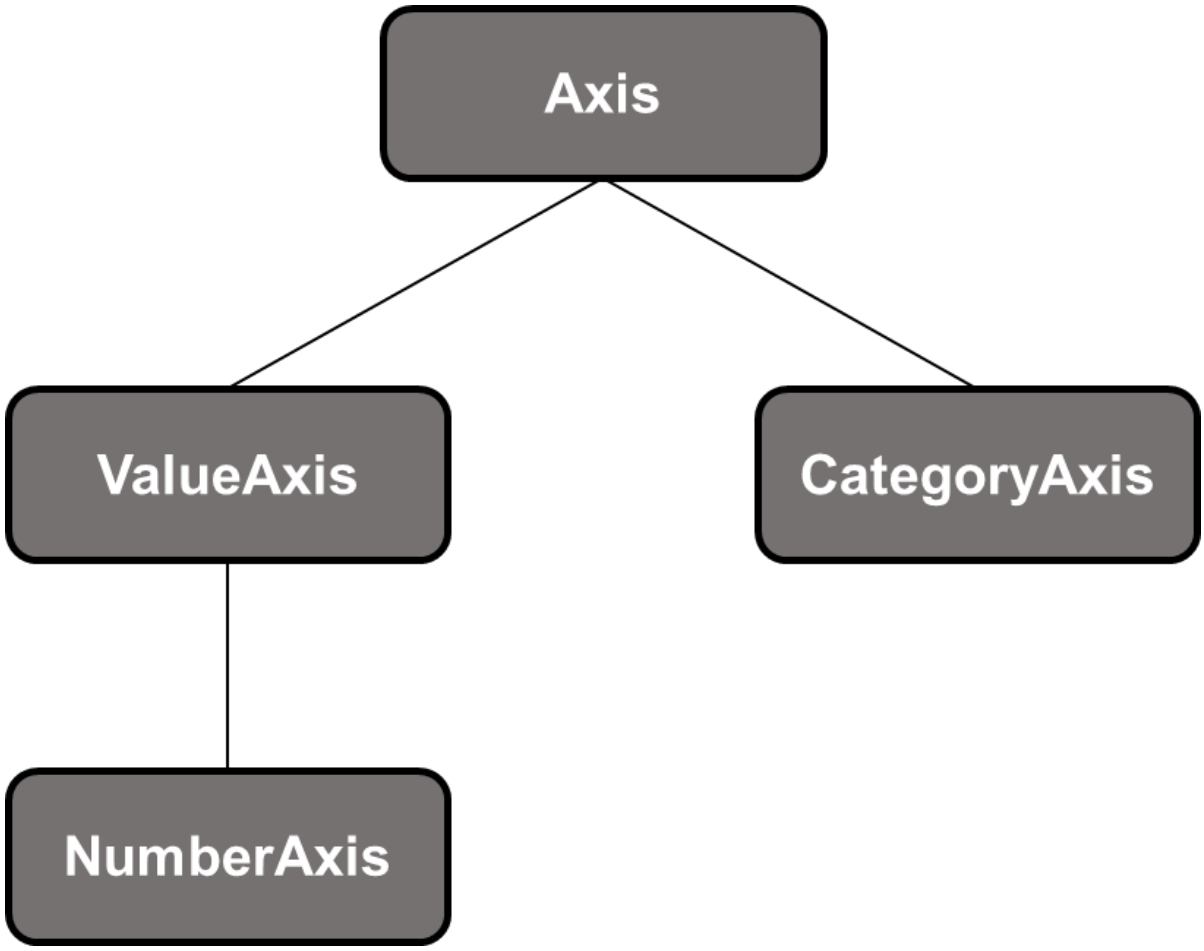
As observed in the above code, while instantiating, you need to pass two objects representing the X and Y axis of the chart respectively.

Defining the Axis

In general, the axis of the charts can be represented by:

- Numbers such as Population, Age and
- Categories such as Days in a Week, Countries.

In JavaFX, an axis is an abstract class representing X or Y axis. It has two subclasses to define each type of axis, namely **CategoryAxis** and **NumberAxis** as shown in the following diagram –



CategoryAxis: By instantiating this class, you can define (create) an X or Y axis along which each value represents a category. You can define a Category axis by instantiating this class as shown below –

```
CategoryAxis xAxis = new CategoryAxis();
```

To this axis, you need set the list of categories and label to the axis as shown below –

```
//setting the list of categories.
xAxis.setCategories(FXCollections.<String>observableArrayList(Arrays.asList("name1", "name2"….)));


//Setting label to the axis
xAxis.setLabel("name of the axis ");
```

NumberAxis: By instantiating this class, you can define (create) an X or Y axis along which each value represents a Numerical value. You can use any Number type with this **Axis**, **Long**, **Double**, **BigDecimal**, etc. You can define a Number axis by instantiating this class as follows:

```
//Defining the axis
NumberAxis yAxis = new NumberAxis();


//Setting labelto the axis
yAxis.setLabel("name of the axis");
```

Passing Data to XY Charts

All the XY charts are represented along the XY plane. To plot a set of points in a chart, we need to specify a series of XY coordinates.

The **<X,Y>** class of the **javafx.scene.chart** package is a class using which, you can send data to a chart. This class holds an observable list of named series. You can get this list using the **getData()** method of **XYChart.Series** class as shown below **–**

```
ObservableList list = series.getData();
```

Where, **series** is the object of the **XYChart.Series** class. You can add data to this list using the **add()** method as follows –

```
list.add(new XYChart.Data(x-axis data, y-axis data));
```

These two lines can be written together as shown below –

```
series.getData().add(new XYChart.Data(x-axis data, y-axis data));
```

The following table gives a description of various charts (classes) provided by JavaFX –

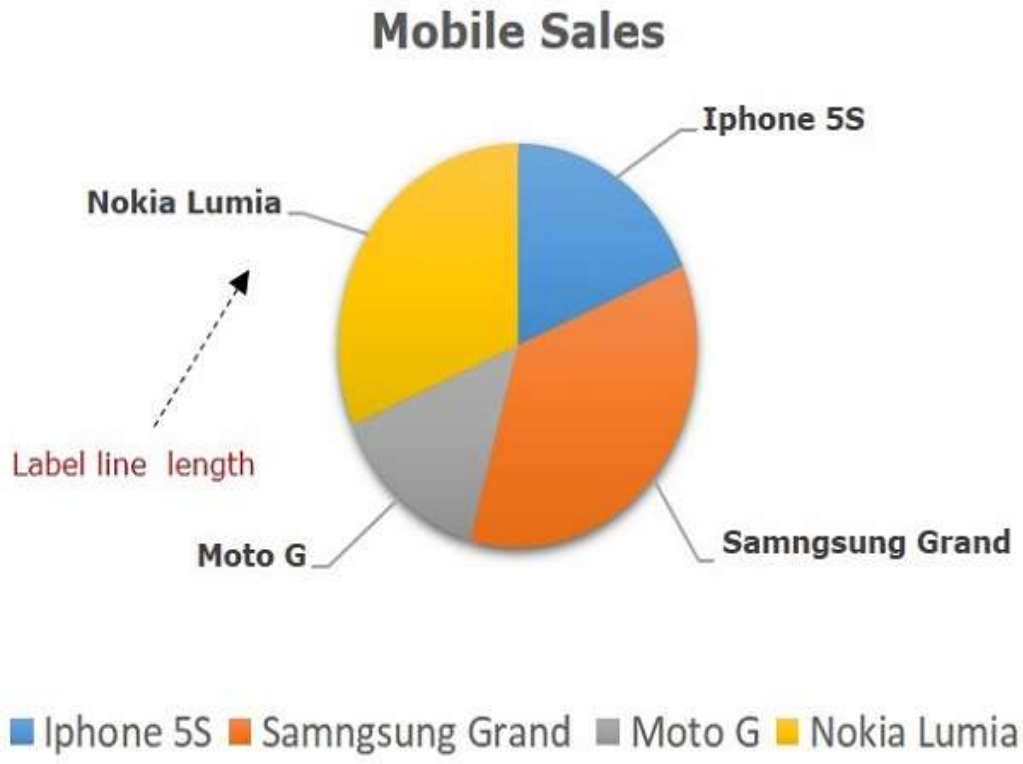
S. No.	Chart and Description
1	**Pie Chart:** A pie-chart is a representation of values as slices of a circle with different colors. These slices are labeled and the values corresponding to each slice is represented in the chart. In JavaFX, a pie chart is represented by a class named **PieChart.** This class belongs to the package **javafx.scene.chart.**
2	**Line Chart:** A line chart or line graph displays information as a series of data points (markers) connected by straight line segments. Line Chart shows how the data changes at equal time frequency. In JavaFX, a line chart is represented by a class named **LineChart.** This class belongs to the package **javafx.scene.chart.** By instantiating this class, you can create a LineChart node in JavaFX.
3	**Area Chart**: Area charts are used to draw area based charts. It plots the area between the given series of points and the axis. In general, this chart is used to compare two quantities. In JavaFX, an Area chart is represented by a class named **AreaChart.** This class belongs to the package **javafx.scene.chart.** By instantiating this class, you can create a AreaChart node in JavaFX.
4	**Bar Chart:** A bar chart is used to represent grouped data using rectangular bars. The length of these bars depicts the values. The bars in the bar chart can be plotted vertically or horizontally. In JavaFX, a Bar chart is represented by a class named **BarChart.** This class belongs to the package **javafx.scene.chart.** By instantiating this class, you can create a BarChart node in JavaFX.
5	**Bubble Chart:** A bubble chart is used to plat three-dimensional data. The third dimension will be represented by the size (radius) of the bubble. In JavaFX, a Bubble chart is represented by a class named **BubbleChart.** This class belongs to the package **javafx.scene.chart.** By instantiating this class, you can create a BubbleChart node in JavaFX.

6	**Scatter Chart:** A scatterplot is a type of graph which uses values from two variables plotted in a Cartesian plane. It is usually used to find out the relationship between two variables. In JavaFX, a Scatter chart is represented by a class named **ScatterChart.** This class belongs to the package **javafx.scene.chart.** By instantiating this class, you can create a ScatterChart node in JavaFX.
7	**Stacked Area Chart:** In JavaFX, a Stacked Area chart is represented by a class named **StackedAreaChart.** This class belongs to the package **javafx.scene.chart.** By instantiating this class, you can create an StackedAreaChart node in JavaFX.
8	**Stacked Bar Chart**: In JavaFX, a Stacked Bar chart is represented by a class named **StackedBarChart.** This class belongs to the package **javafx.scene.chart.** By instantiating this class, you can create a StackedBarChart node in JavaFX.

Charts – Pie Chart

A pie-chart is a representation of values as slices of a circle with different colors. These slices are labeled and the values corresponding to each slice is represented in the chart.

Following is a Pie Chart depicting the mobile sales of various companies at an instance.



In JavaFX, a pie chart is represented by a class named **PieChart.** This class belongs to the package **javafx.scene.chart.**

By instantiating this class, you can create a PieChart node in JavaFX.

This class has 5 properties which are as follows –

- **clockwise:** This is a Boolean Operator; on setting this operator true, the data slices in the pie charts will be arranged clockwise starting from the start angle of the pie chart.
- **data:** This represents an **ObservableList** object, which holds the data of the pie chart.
- **labelLineLength:** An integer operator representing the length of the lines connecting the labels and the slices of the pie chart.
- **labelsVisible:** This is a Boolean Operator; on setting this operator true, the labels for the pie charts will be drawn. By default, this operator is set to be true.
- **startAngle**: This is a double type operator, which represents the angle to start the first pie slice at.

To generate a pie chart, prepare an **ObservableList** object as shown in the following code block –

```
//Preparing ObservbleList object
ObservableList<PieChart.Data> pieChartData =
   FXCollections.observableArrayList(
      new PieChart.Data("Iphone 5S", 13),
      new PieChart.Data("Samsung Grand", 25),
      new PieChart.Data("MOTO G", 10),
      new PieChart.Data("Nokia Lumia", 22));
```

After preparing the **ObservableList** object, pass it as an argument to the constructor of the class **PieChart** as follows –

```
//Creating a Pie chart
PieChart pieChart = new PieChart(pieChartData);
```

Or, by using the method named **setData()** of the class named **PieChart** of the package named **javafx.scene.chart.**

```
pieChart.setData(pieChartData);
```

To generate a **PieChart** in JavaFX, follow the steps given below.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application** and implement the **start()** method of this class as follows.

```
public class ClassName extends Application {


    @Override
    public void start(Stage primaryStage) throws Exception {
    }
}
```

Step 2: Preparing the ObservableList Object

Prepare an object of the interface **ObservableList** object by passing the data of the pie chart as shown below –

```
ObservableList<PieChart.Data> pieChartData =
      FXCollections.observableArrayList(
         new PieChart.Data("Iphone 5S", 13),
         new PieChart.Data("Samsung Grand", 25),
         new PieChart.Data("MOTO G", 10),
         new PieChart.Data("Nokia Lumia", 22));
```

Step 3: Creating a PieChart Object

Create a **PieChart** by passing the **ObservableList** object as shown below.

```
//Creating a Pie chart
 PieChart pieChart = new PieChart(pieChartData);
```

Step 4: Setting the Title of the Pie Chart

Set the title of the Pie Chart using the **setTitle()** method of the class **PieChart**. This belongs to the package **javafx.scene.chart** –

```
//Setting the title of the Pie chart
 pieChart.setTitle("Mobile Sales");
```

Step 5: Setting the Slices Clockwise

Set the slices of the Pie Charts clockwise. This is done by passing Boolean value true to the **setClockwise()** method of the class **PieChart**. This belongs to the package **javafx.scene.chart** –

```
//setting the direction to arrange the data
 pieChart.setClockwise(true);
```

Step 6: Set the Length of the Label Line

Set the length of the label line using the **setLabelLineLength()** method of the class **PieChart** which belongs to the package **javafx.scene.chart,** as follows –

```
//Setting the length of the label line
pieChart.setLabelLineLength(50);
```

Step 7: Set the Labels Visible

Set the labels of the pie chart to visible by passing the Boolean value **true** to the method **setLabelsVisible()** of the class **PieChart**. This belongs to the package **javafx.scene.chart** –

```
//Setting the labels of the pie chart visible
pieChart.setLabelsVisible(true);
```

Step 8: Set the Start Angle of the Pie Chart

Set the Start angle of the pie chart using the **setStartAngle()** method of the class **PieChart**. This belongs to the package **javafx.scene.chart** –

```
//Setting the start angle of the pie chart
pieChart.setStartAngle(180);
```

Step 9: Creating a Group Object

In the **start()** method, create a group object by instantiating the class named **Group**. This belongs to the package **javafx.scene**.

Pass the PieChart (node) object, created in the previous step as a parameter to the constructor of the Group class. This should be done in order to add it to the group as follows –

```
Group root = new Group(piechart);
```

Step 10: Creating a Scene Object

Create a Scene by instantiating the class named **Scene**, which belongs to the package **javafx.scene.** To this class, pass the Group object (**root**) created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen, along with the object of the Group class as shown below.

```
Scene scene = new Scene(group ,600, 300);
```

Step 11: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage Object, which is passed to the start method of the scene class as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 12: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using this method as shown below.

```
primaryStage.setScene(scene);
```

Step 13: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 14: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows.

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

Example

The table given below depicts mobile sale with the help of a pie chart. The following table has a list of different mobile brands and their sale (units per day).

S.No	Mobile Brands	Sales (Units per day)
1	Iphone 5S	20
2	Samsung Grand	20
3	MOTO G	40
4	Nokia Lumia	10

Following is a Java program which generates a pie chart, depicting the above data using JavaFX. Save this code in a file with the name **PieChartExample.java.**

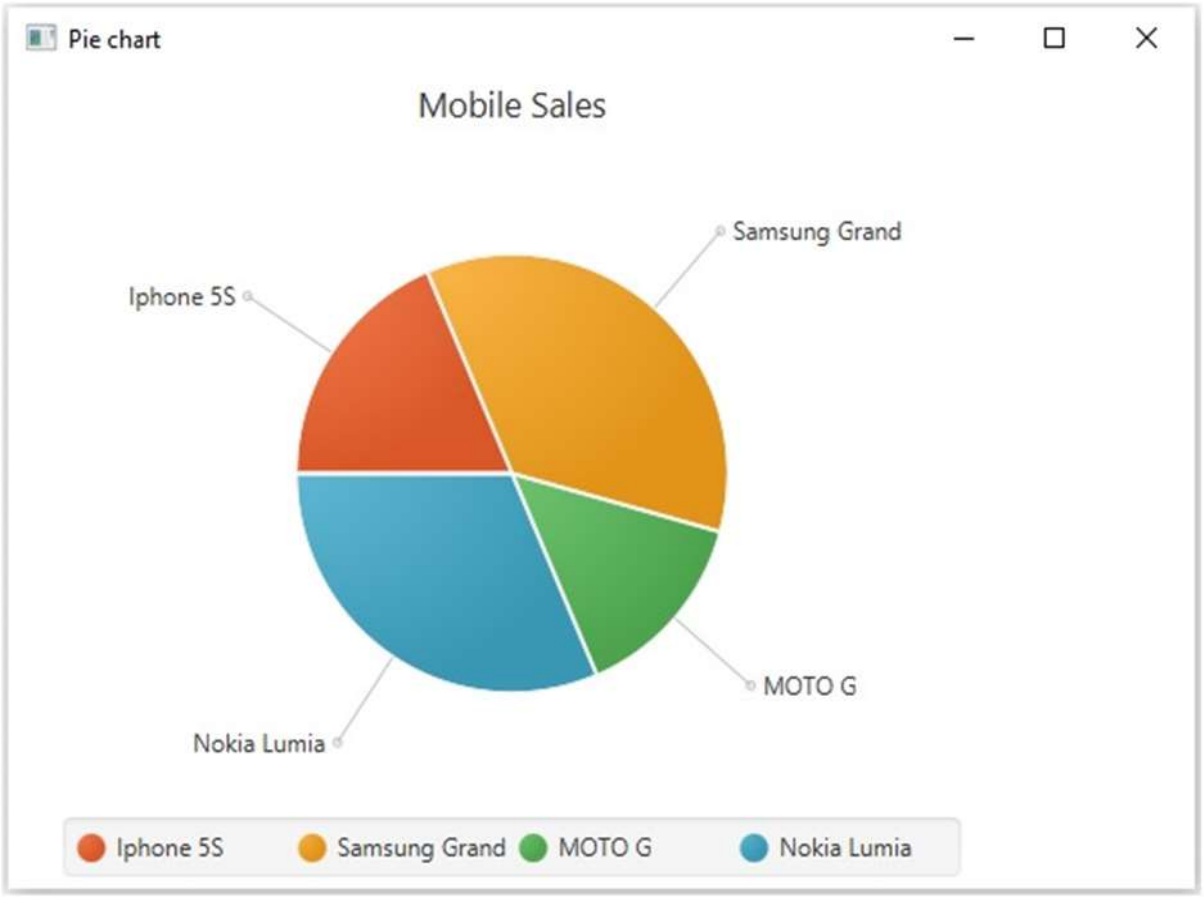
```
import javafx.application.Application;
import javafx.collections.FXCollections;

import javafx.collections.ObservableList;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.chart.PieChart;

public class PieChartExample extends Application {

   @Override
   public void start(Stage stage) {

      //Preparing ObservbleList object
      ObservableList<PieChart.Data> pieChartData =
      FXCollections.observableArrayList(
         new PieChart.Data("Iphone 5S", 13),
         new PieChart.Data("Samsung Grand", 25),
         new PieChart.Data("MOTO G", 10),
         new PieChart.Data("Nokia Lumia", 22));

       //Creating a Pie chart
       PieChart pieChart = new PieChart(pieChartData);

       //Setting the title of the Pie chart
       pieChart.setTitle("Mobile Sales");

       //setting the direction to arrange the data
       pieChart.setClockwise(true);

      //Setting the length of the label line
       pieChart.setLabelLineLength(50);
```

```
        //Setting the labels of the pie chart visible
        pieChart.setLabelsVisible(true);

        //Setting the start angle of the pie chart

        pieChart.setStartAngle(180);

        //Creating a Group object
        Group root = new Group(pieChart);

        //Creating a scene object
        Scene scene = new Scene(root, 600, 400);

        //Setting title to the Stage
        stage.setTitle("Pie chart");

        //Adding scene to the satge
        stage.setScene(scene);

        //Displaying the contents of the stage
        stage.show();

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac PieChartExample.java
java PieChartExample
```

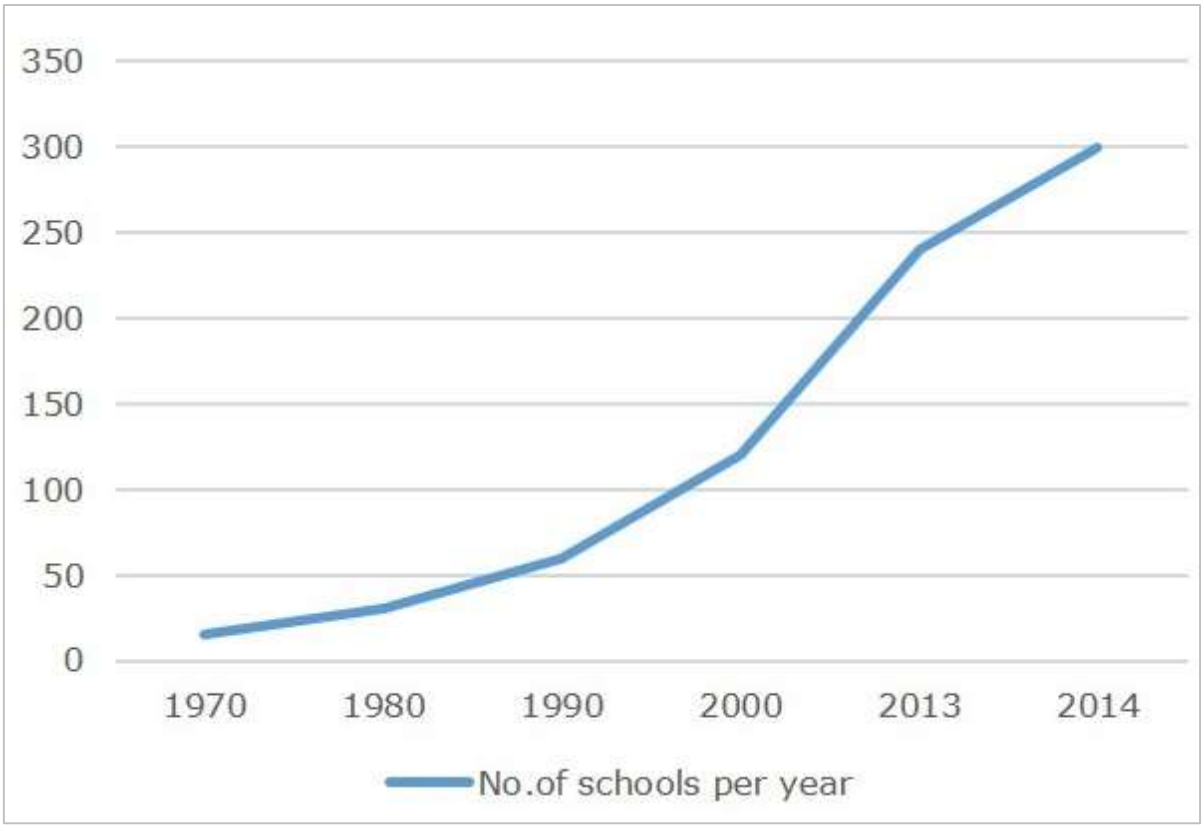
On executing, the above program generates a JavaFX window displaying a pie chart as shown below.



Charts – Line Chart

A line chart or line graph displays information as a series of data points (markers) connected by straight line segments. A Line Chart shows how the data changes at equal time frequency.

Following is a Line chart depicting the number of schools in different years.



In JavaFX, a line chart is represented by a class named **LineChart.** This class belongs to the package **javafx.scene.chart.** By instantiating this class, you can create a LineChart node in JavaFX.

To generate a line chart in JavaFX, you should follow the steps given below.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application**. You can then implement the **start()** method of this class as follows.

```
public class ClassName extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
    }
}
```

Step 2: Defining the axis

Define the X and Y axis of the line chart and set labels to them.

In our example, the X axis represent the years starting from 1960 to 2020 having major tick mark at every ten years.

```
//Defining X axis
NumberAxis xAxis = new NumberAxis(1960, 2020, 10);
xAxis.setLabel("Years");

//Defining y axis
NumberAxis yAxis = new NumberAxis(0, 350, 50);
yAxis.setLabel("No.of schools");
```

Step 3: Creating the Line Chart

Create a line chart by instantiating the class named **LineChart** of the package **javafx.scene.chart**. To the constructor of this class, pass the objects representing the X and Y axis created in the previous step.

```
LineChart linechart = new LineChart(xAxis, yAxis);
```

Step 4: Preparing the Data

Instantiate the **XYChart.Series** class. Then add the data (a series of, x and y coordinates) to the Observable list of this class as follows –

```
XYChart.Series series = new XYChart.Series();
series.setName("No of schools in an year");

series.getData().add(new XYChart.Data(1970, 15));
series.getData().add(new XYChart.Data(1980, 30));
series.getData().add(new XYChart.Data(1990, 60));
series.getData().add(new XYChart.Data(2000, 120));
series.getData().add(new XYChart.Data(2013, 240));
series.getData().add(new XYChart.Data(2014, 300));
```

Step 5: Add Data to the Line Chart

Add the data series prepared in the previous step to the line chart as follows –

```
//Setting the data to Line chart
linechart.getData().add(series);
```

Step 6: Creating a Group Object

In the **start()** method, create a group object by instantiating the class named **Group**. This belongs to the package **javafx.scene**.

Pass the LineChart (node) object, created in the previous step as a parameter to the constructor of the Group class. This should be done in order to add it to the group as follows –

```
Group root = new Group(linechart);
```

Step 7: Creating a Scene Object

Create a Scene by instantiating the class named **Scene**, which belongs to the package **javafx.scene.** To this class, pass the Group object (**root**), created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 8: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object, which is passed to the start method of the scene class as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 9: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using this method as follows.

```
primaryStage.setScene(scene);
```

Step 10: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 11: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows.

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

Example

The following table depicts the number of schools that were in an area from the year 1970 to 2014.

Year	Number of Schools
1970	15
1980	30
1990	60
2000	120
2013	240
2014	300

Following is a Java program which generates a line chart, depicting the above data, using JavaFX.

Save this code in a file with the name **LineChartExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.chart.LineChart;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;

public class LineChartExample extends Application {



   @Override
   public void start(Stage stage) {
```

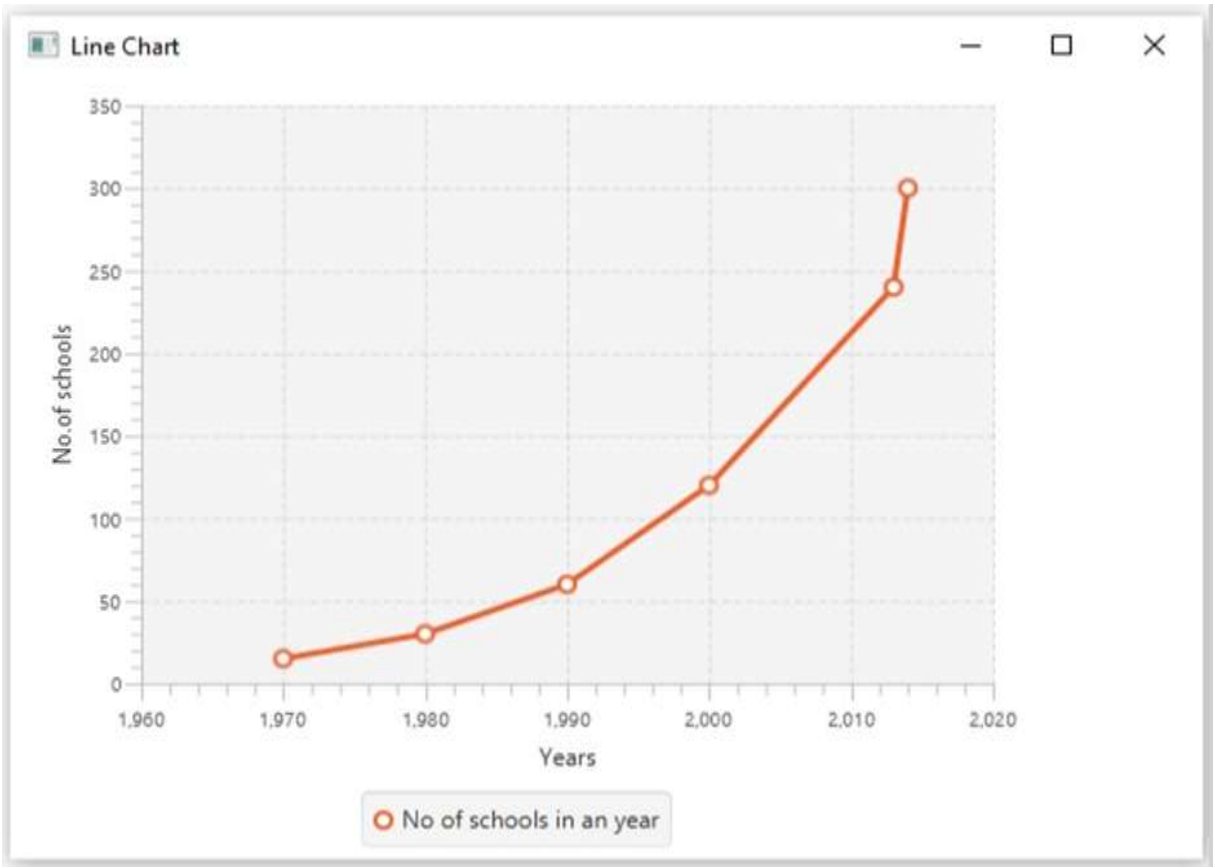
```
        //Defining the x axis
        NumberAxis xAxis = new NumberAxis(1960, 2020, 10);
        xAxis.setLabel("Years");

        //Defining the y axis
        NumberAxis yAxis = new NumberAxis(0, 350, 50);
        yAxis.setLabel("No.of schools");

        //Creating the line chart
        LineChart linechart = new LineChart(xAxis, yAxis);

        //Prepare XYChart.Series objects by setting data
        XYChart.Series series = new XYChart.Series();
        series.setName("No of schools in an year");

        series.getData().add(new XYChart.Data(1970, 15));
        series.getData().add(new XYChart.Data(1980, 30));
        series.getData().add(new XYChart.Data(1990, 60));
        series.getData().add(new XYChart.Data(2000, 120));
        series.getData().add(new XYChart.Data(2013, 240));
        series.getData().add(new XYChart.Data(2014, 300));

        //Setting the data to Line chart
        linechart.getData().add(series);

        //Creating a Group object
        Group root = new Group(linechart);

        //Creating a scene object
        Scene scene = new Scene(root, 600, 400);

        //Setting title to the Stage
        stage.setTitle("Line Chart");

        //Adding scene to the satge
        stage.setScene(scene);
```

```
        //Displaying the contents of the stage
        stage.show();
    }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac LineChartExample.java
java LineChartExample
```

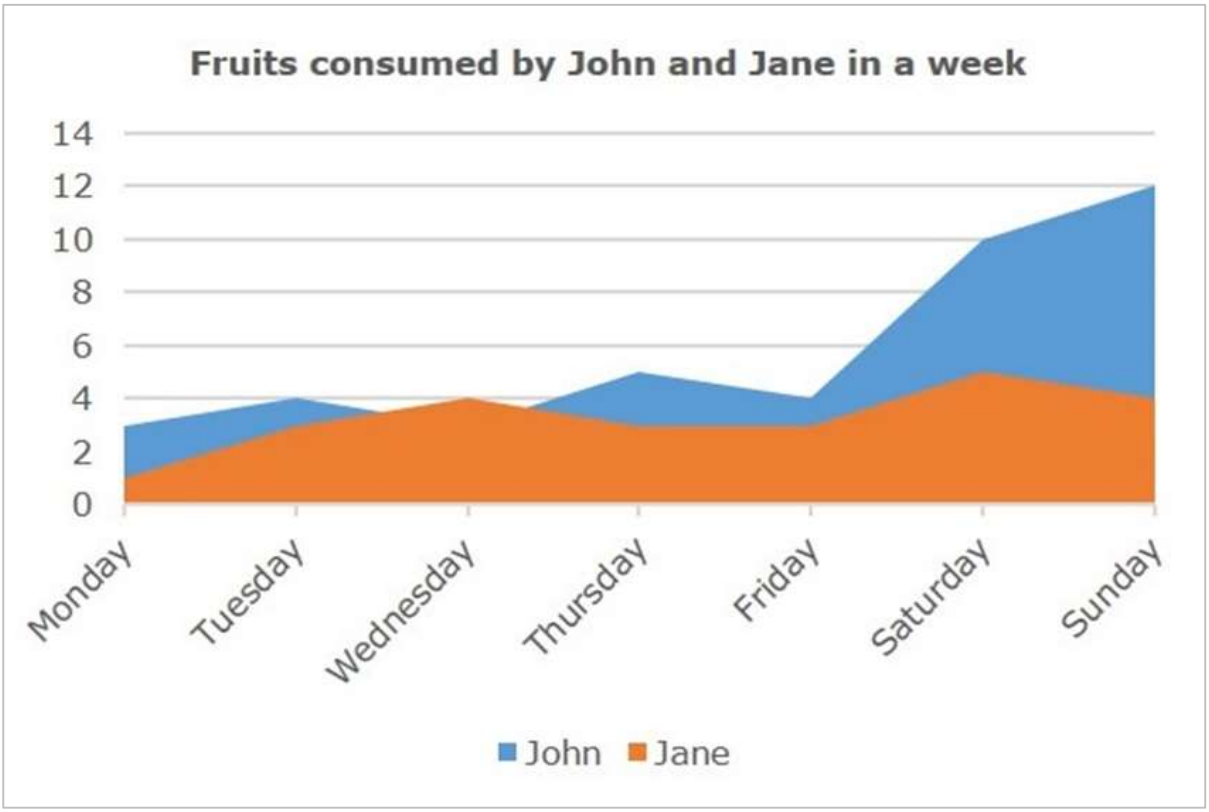
On executing, the above program generates a JavaFX window displaying a line chart as shown below.



Charts – Area Chart

Area charts are used to draw area based charts. It plots the area between the given series of points and the axis. In general, this chart is used to compare two quantities.

Following is an Area chart depicting the number of fruits consumed by two people in a week.



In JavaFX, an Area chart is represented by a class named **AreaChart.** This class belongs to the package **javafx.scene.chart.** By instantiating this class, you can create an AreaChart node in JavaFX.

To generate an area chart in JavaFX, follow the steps given below.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application** and implement the **start()** method of this class as follows.

```
public class ClassName extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
    }
}
```

Step 2: Defining the Axis

Define the X and Y axis of the area chart and set labels to them. In our example, X axis represents the days in a week and the y axis represents the units of fruits consumed.

```
//Defining the X axis
CategoryAxis xAxis = new CategoryAxis();

//Defining the y Axis
NumberAxis yAxis = new NumberAxis(0, 15, 2.5);
yAxis.setLabel("Fruit units");
```

Step 3: Creating the Area Chart

Create a line chart by instantiating the class named **AreaChart** of the package **javafx.scene.chart**. To the constructor of this class, pass the objects representing the X and Y axis created in the previous step.

```
//Creating the Area chart
AreaChart<String, Number> areaChart = new AreaChart(xAxis, yAxis);
areaChart.setTitle("Average fruit consumption during one week");
```

Step 4: Preparing the data

Instantiate the **XYChart.Series** class. Then add the data (a series of, x and y coordinates) to the Observable list of this class as follows –

```
//Prepare XYChart.Series objects by setting data
XYChart.Series series1 = new XYChart.Series();
series1.setName("John");
series1.getData().add(new XYChart.Data("Monday", 3));
series1.getData().add(new XYChart.Data("Tuesday", 4));
series1.getData().add(new XYChart.Data("Wednesday", 3));
series1.getData().add(new XYChart.Data("Thursday", 5));
series1.getData().add(new XYChart.Data("Friday", 4));
series1.getData().add(new XYChart.Data("Saturday", 10));
series1.getData().add(new XYChart.Data("Sunday", 12));

XYChart.Series series2= new XYChart.Series();
series2.setName("Jane");
series2.getData().add(new XYChart.Data("Monday", 1));
series2.getData().add(new XYChart.Data("Tuesday", 3));
```

```
series2.getData().add(new XYChart.Data("Wednesday", 4));

series2.getData().add(new XYChart.Data("Thursday", 3));
series2.getData().add(new XYChart.Data("Friday", 3));
series2.getData().add(new XYChart.Data("Saturday", 5));
series2.getData().add(new XYChart.Data("Sunday", 4));
```

Step 5: Add Data to the Area Chart

Add the data series prepared in the previous step to the area chart as follows –

```
//Setting the XYChart.Series objects to area chart
areaChart.getData().addAll(series1,series2);
```

Step 6: Creating a Group Object

In the **start()** method, create a group object by instantiating the class named **Group**, which belongs to the package **javafx.scene**.

Pass the AreaChart (node) object, created in the previous step as a parameter to the constructor of the Group class. This should be done in order to add it to the group as follows –

```
Group root = new Group(areaChart);
```

Step 7: Creating a Scene Object

Create a Scene by instantiating the class named **Scene,** which belongs to the package **javafx.scene.** To this class, pass the Group object (**root**) created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen, along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 8: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object, which is passed to the start method of the scene class as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 9: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using the following method.

```
primaryStage.setScene(scene);
```

Step 10: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 11: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows.

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

Example

The following table depicts the number of fruits consumed by John and Jane in a week.

Day of the week	Fruits consumed by John	Fruits consumed by Jane
Monday	3	1
Tuesday	4	3
Wednesday	3	4
Thursday	5	3
Friday	4	3
Saturday	10	5
Sunday	12	4

Following is a Java program which generates an area chart, depicting the above data using JavaFX.

Save this code in a file with the name **AreaChartExample.java.**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.chart.AreaChart;
import javafx.scene.chart.CategoryAxis;
import javafx.stage.Stage;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;

public class AreaChartExample extends Application {

   @Override
   public void start(Stage stage) {

        //Definging the X axis
        CategoryAxis xAxis = new CategoryAxis();

        //defining the y Axis
        NumberAxis yAxis = new NumberAxis(0, 15, 2.5);
        yAxis.setLabel("Fruit units");

        //Creating the Area chart
        AreaChart<String, Number> areaChart = new AreaChart(xAxis, yAxis);
        areaChart.setTitle("Average fruit consumption during one week");

        //Prepare XYChart.Series objects by setting data
        XYChart.Series series1 = new XYChart.Series();
        series1.setName("John");
        series1.getData().add(new XYChart.Data("Monday", 3));
        series1.getData().add(new XYChart.Data("Tuesday", 4));
        series1.getData().add(new XYChart.Data("Wednesday", 3));
        series1.getData().add(new XYChart.Data("Thursday", 5));
        series1.getData().add(new XYChart.Data("Friday", 4));
        series1.getData().add(new XYChart.Data("Saturday", 10));
```

```
        series1.getData().add(new XYChart.Data("Sunday", 12));

        XYChart.Series series2= new XYChart.Series();
        series2.setName("Jane");
        series2.getData().add(new XYChart.Data("Monday", 1));
        series2.getData().add(new XYChart.Data("Tuesday", 3));
        series2.getData().add(new XYChart.Data("Wednesday", 4));
        series2.getData().add(new XYChart.Data("Thursday", 3));
        series2.getData().add(new XYChart.Data("Friday", 3));
        series2.getData().add(new XYChart.Data("Saturday", 5));
        series2.getData().add(new XYChart.Data("Sunday", 4));

        //Setting the XYChart.Series objects to area chart
        areaChart.getData().addAll(series1,series2);

        //Creating a Group object
        Group root = new Group(areaChart);

        //Creating a scene object
        Scene scene = new Scene(root, 600, 400);

        //Setting title to the Stage
        stage.setTitle("Area Chart");

        //Adding scene to the satge
        stage.setScene(scene);

        //Displaying the contents of the stage
        stage.show();

    }

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

Compile and execute the saved java file from the command prompt using the following commands.

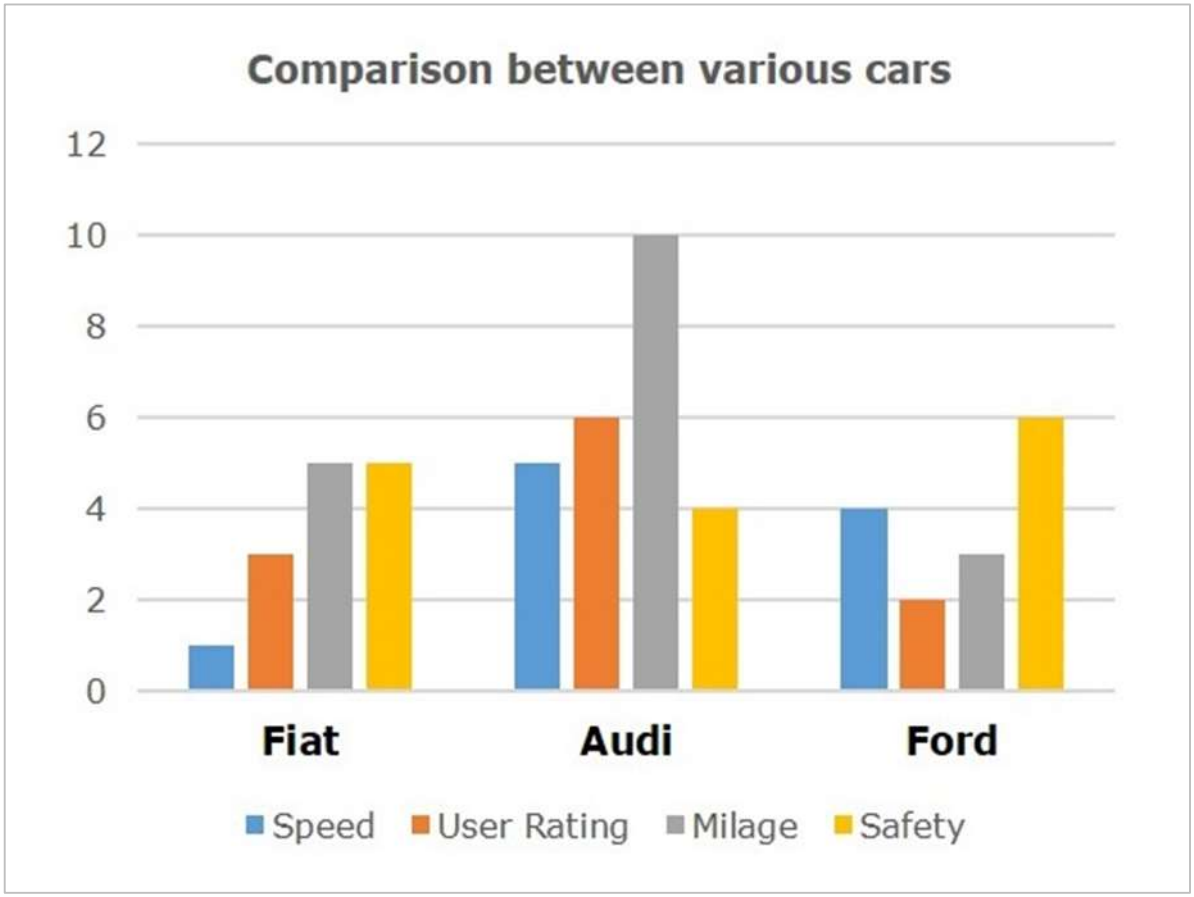
```
javac AreaChartExample.java

java AreaChartExample
```

On executing, the above program generates a JavaFX window displaying an area chart as shown below.

Charts – Bar Chart

A bar chart is used to represent grouped data using rectangular bars. The length of these bars depicts the values. The bars in the bar chart can be plotted vertically or horizontally.

Following is a bar chart, comparing various car brands.



In JavaFX, a Bar chart is represented by a class named **BarChart.** This class belongs to the package **javafx.scene.chart.** By instantiating this class, you can create an BarChart node in JavaFX.

To generate a bar chart in JavaFX, follow the steps given below.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application**. You can then implement the **start()** method of this class as follows.

```
public class ClassName extends Application {


    @Override
    public void start(Stage primaryStage) throws Exception {


    }


}
```

Step 2: Defining the Axis

Define the X and Y axis of the bar chart and set labels to them. In our example, X axis represents the category of comparison and the y axis represents the score.

```
//Defining the x axis
CategoryAxis xAxis = new CategoryAxis();

xAxis.setCategories(FXCollections.<String>observableArrayList(Arrays.asList("S
peed", "User rating", "Milage", "Safety")));
xAxis.setLabel("category");


//Defining the y axis
NumberAxis yAxis = new NumberAxis();
yAxis.setLabel("score");
```

Step 3: Creating the Bar Chart

Create a line chart by instantiating the class named **BarChart** of the package **javafx.scene.chart**. To the constructor of this class, pass the objects representing the X and Y axis created in the previous step.

```
//Creating the Bar chart
BarChart<String, Number> barChart = new BarChart<>(xAxis, yAxis);
barChart.setTitle("Comparison between various cars");
```

Step 4: Preparing the Data

Instantiate the **XYChart.Series** class and add the data (a series of, x and y coordinates) to the Observable list of this class as follows –

```
//Prepare XYChart.Series objects by setting data
XYChart.Series<String, Number> series1 = new XYChart.Series<>();
series1.setName("Fiat");
series1.getData().add(new XYChart.Data<>("Speed", 1.0));
series1.getData().add(new XYChart.Data<>("User rating", 3.0));
series1.getData().add(new XYChart.Data<>("Milage", 5.0));
series1.getData().add(new XYChart.Data<>("Safety", 5.0));



XYChart.Series<String, Number> series2 = new XYChart.Series<>();
series2.setName("Audi");
```

```
series2.getData().add(new XYChart.Data<>("Speed", 5.0));
series2.getData().add(new XYChart.Data<>("User rating", 6.0));

series2.getData().add(new XYChart.Data<>("Milage", 10.0));
series2.getData().add(new XYChart.Data<>("Safety", 4.0));

XYChart.Series<String, Number> series3 = new XYChart.Series<>();
series3.setName("Ford");
series3.getData().add(new XYChart.Data<>("Speed", 4.0));
series3.getData().add(new XYChart.Data<>("User rating", 2.0));
series3.getData().add(new XYChart.Data<>("Milage", 3.0));
series3.getData().add(new XYChart.Data<>("Safety", 6.0));
```

Step 5: Add Data to the Bar Chart

Add the data series prepared in the previous step to the bar chart as follows –

```
//Setting the data to bar chart
barChart.getData().addAll(series1, series2, series3);
```

Step 6: Creating a Group Object

In the **start()** method, create a group object by instantiating the class named **Group**. This belongs to the package **javafx.scene**.

Pass the BarChart (node) object, created in the previous step as a parameter to the constructor of the Group class. This should be done in order to add it to the group as follows –

```
Group root = new Group(barChart);
```

Step 7: Creating a Scene Object

Create a Scene by instantiating the class named **Scene**, which belongs to the package **javafx.scene.** To this class, pass the Group object (**root**) created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen, along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 8: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object, which is passed to the start method of the scene class as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 9: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using the following method.

```
primaryStage.setScene(scene);
```

Step 10: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 11: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows.

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

Example

The following example depicts various car statistics with the help of a bar chart. Following is a list of car brands along with their different characteristics, which we will show using a bar chart:

Car	Speed	User Rating	Millage	Safety
Fiat	1.0	3.0	5.0	5.0
Audi	5.0	6.0	10.0	4.0
Ford	4.0	2.0	3.0	6.0

Following is a Java program which generates a bar chart, depicting the above data using JavaFX.

Save this code in a file with the name **BarChartExample.java.**

```
import java.util.Arrays;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.chart.BarChart;
import javafx.scene.chart.CategoryAxis;
import javafx.stage.Stage;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;

public class BarChartExample extends Application {
   @Override
   public void start(Stage stage) {
      //Defining the axes
      CategoryAxis xAxis = new CategoryAxis();
      xAxis.setCategories(FXCollections.<String>
      observableArrayList(Arrays.asList("Speed", "User rating", "Milage",
"Safety")));
      xAxis.setLabel("category");

      NumberAxis yAxis = new NumberAxis();
      yAxis.setLabel("score");

      //Creating the Bar chart
      BarChart<String, Number> barChart = new BarChart<>(xAxis, yAxis);
      barChart.setTitle("Comparison between various cars");

      //Prepare XYChart.Series objects by setting data
      XYChart.Series<String, Number> series1 = new XYChart.Series<>();
      series1.setName("Fiat");
      series1.getData().add(new XYChart.Data<>("Speed", 1.0));
      series1.getData().add(new XYChart.Data<>("User rating", 3.0));
```

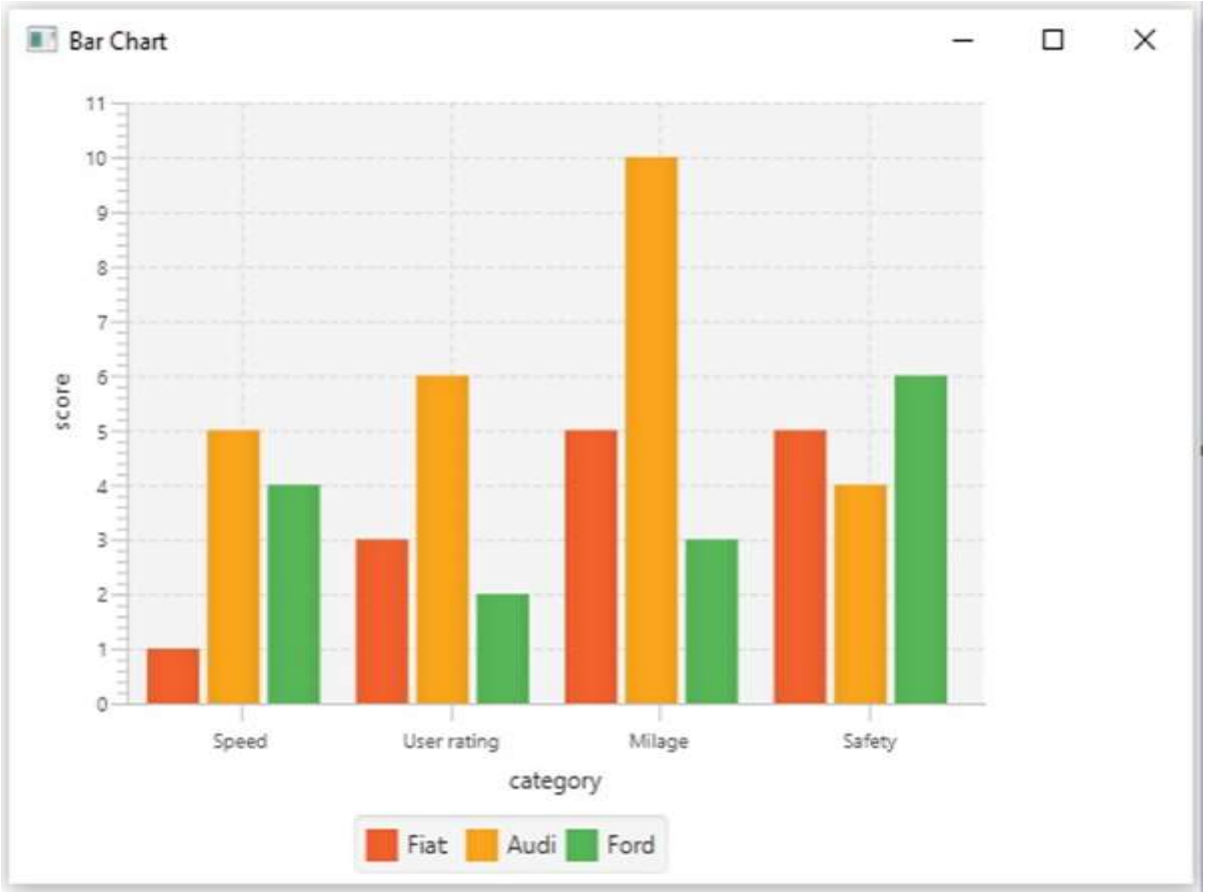
```
      series1.getData().add(new XYChart.Data<>("Milage", 5.0));
      series1.getData().add(new XYChart.Data<>("Safety", 5.0));

      XYChart.Series<String, Number> series2 = new XYChart.Series<>();
      series2.setName("Audi");
      series2.getData().add(new XYChart.Data<>("Speed", 5.0));
      series2.getData().add(new XYChart.Data<>("User rating", 6.0));
      series2.getData().add(new XYChart.Data<>("Milage", 10.0));
      series2.getData().add(new XYChart.Data<>("Safety", 4.0));

      XYChart.Series<String, Number> series3 = new XYChart.Series<>();
      series3.setName("Ford");
      series3.getData().add(new XYChart.Data<>("Speed", 4.0));
      series3.getData().add(new XYChart.Data<>("User rating", 2.0));
      series3.getData().add(new XYChart.Data<>("Milage", 3.0));
      series3.getData().add(new XYChart.Data<>("Safety", 6.0));

      //Setting the data to bar chart
      barChart.getData().addAll(series1, series2, series3);

      //Creating a Group object
      Group root = new Group(barChart);

      //Creating a scene object
      Scene scene = new Scene(root, 600, 400);

      //Setting title to the Stage
      stage.setTitle("Bar Chart");

      //Adding scene to the stage
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }
   public static void main(String args[]){
```

```
        launch(args);
    }
}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac BarChartExample.java
java BarChartExample
```

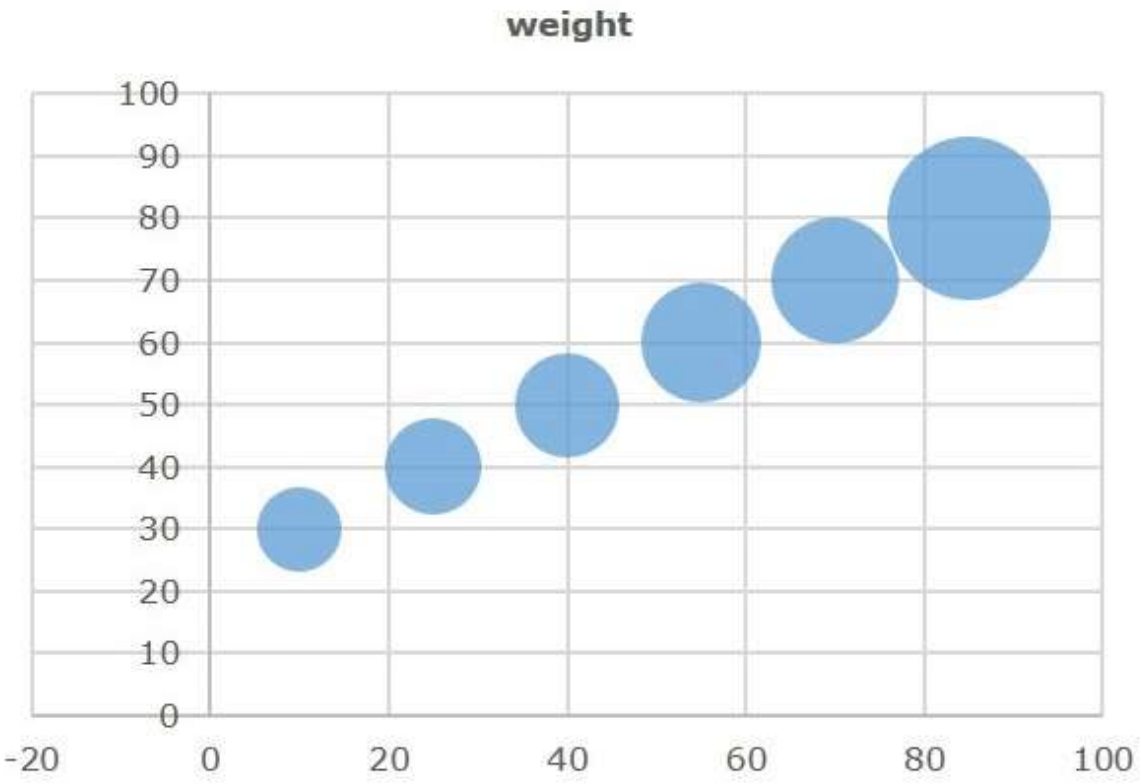
On executing, the above program generates a JavaFX window displaying an area chart as shown below.



Charts – Bubble Chart

A bubble chart is used to plant three-dimensional data; the third dimension will be represented by the size (radius) of the bubble.

The following is a Bubble chart depicting the work done.



In JavaFX, a Bubble chart is represented by a class named **BubbleChart.** This class belongs to the package **javafx.scene.chart.** By instantiating this class, you can create an BubbleChart node in JavaFX.

To generate a bubble chart in JavaFX, follow the steps given below.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application**. You can implement the **start()** method of this class as follows.

```
public class ClassName extends Application {


    @Override
    public void start(Stage primaryStage) throws Exception {
    }
 }
```

Step 2: Defining the Axis

Define the X and Y axis of the bubble chart and set labels to them. In our example, X axis represents the age, Y axis represents the weight. While, the radius of the bubble represents the work done.

```
//Defining the X axis
NumberAxis xAxis = new NumberAxis(0, 100, 10);
xAxis.setLabel("Age");

//Defining Y axis
NumberAxis yAxis = new NumberAxis(20, 100, 10);
yAxis.setLabel("Weight");
```

Step 3: Creating the Bubble Chart

Create a line chart by instantiating the class named **BubbleChart** of the package **javafx.scene.chart**. To the constructor of this class, pass the objects representing the X and Y axis created in the previous step.

```
//Creating the Bubble chart
BubbleChart bubbleChart = new BubbleChart(xAxis, yAxis);
```

Step 4: Preparing the Data

Instantiate the **XYChart.Series** class and add the data (a series of, x and y coordinates) to the Observable list of this class as follows –

```
//Prepare XYChart.Series objects by setting data
XYChart.Series series = new XYChart.Series();
series.setName("work");

series.getData().add(new XYChart.Data(10,30,4));
series.getData().add(new XYChart.Data(25,40,5));
series.getData().add(new XYChart.Data(40,50,9));
series.getData().add(new XYChart.Data(55,60,7));
series.getData().add(new XYChart.Data(70,70,9));
series.getData().add(new XYChart.Data(85,80,6));
```

Step 5: Add Data to the Bubble Chart

Add the data series prepared in the previous step to the area chart as follows –

```
//Setting the data to bar chart
bubbleChart.getData().add(series);
```

Step 6: Creating a Group Object

In the **start()** method, create a group object by instantiating the class named **Group**. This belongs to the package **javafx.scene**.

Pass the BubbleChart (node) object, created in the previous step as a parameter to the constructor of the Group class. This should be done in order to add it to the group as follows -

```
Group root = new Group(bubbleChart);
```

Step 7: Creating a Scene Object

Create a Scene by instantiating the class named **Scene**, which belongs to the package **javafx.scene.** To this class, pass the Group object (**root**), created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen, along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 8: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object, which is passed to the start method of the scene class as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 9: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using the following method.

```
primaryStage.setScene(scene);
```

Step 10: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 11: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows.

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

Example

Let us consider different persons along with their age, weight and work capacities. The work capacity can be treated as the number of hours that is plotted as bubbles in the chart.

WEIGHT								
AGE		30	40	50	60	70	80	
	10	4						WORK
	25		5					
	40			6				
	55				8			
	70					9		
	85						15	

Following is a Java program which generates a bubble chart, depicting the above data using JavaFX.

Save this code in a file with the name **BubbleChartExample.java.**

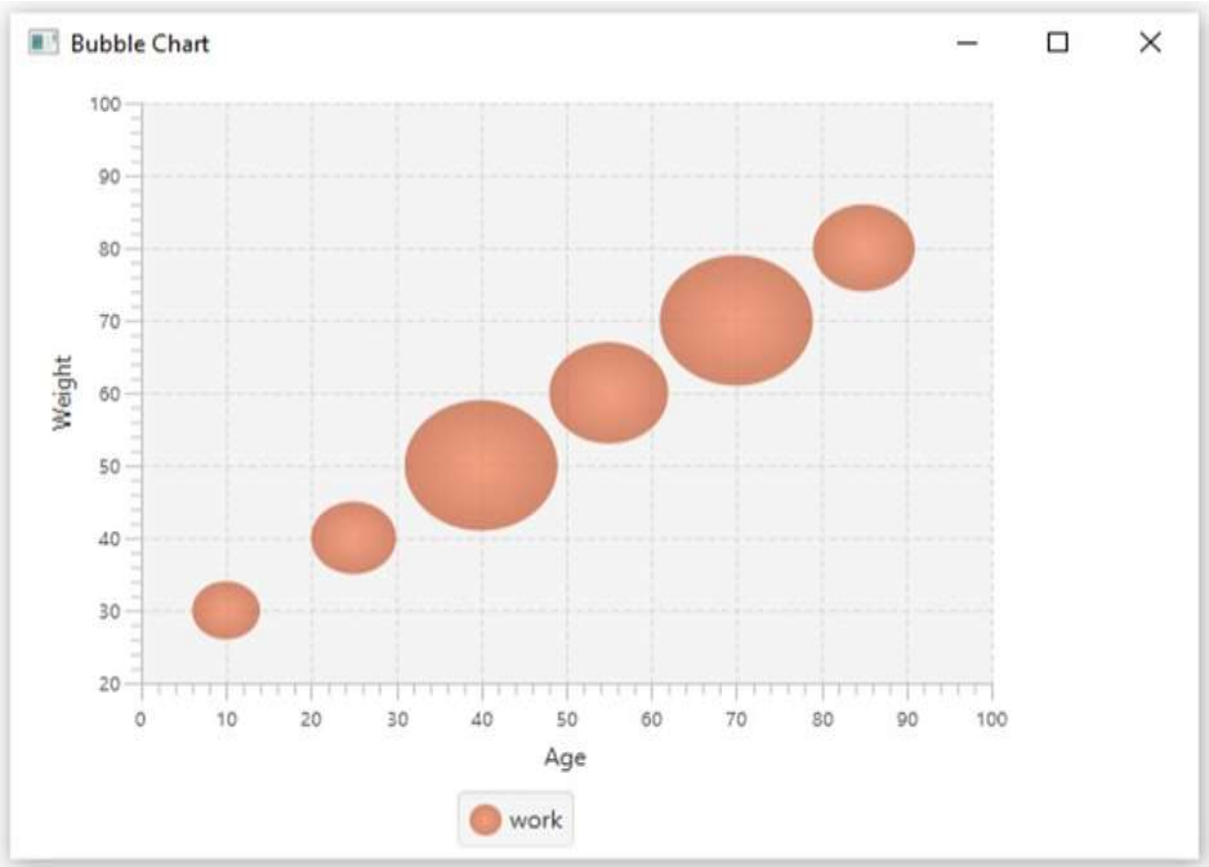
```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.chart.BubbleChart;
import javafx.stage.Stage;

import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;

public class BubbleChartExample extends Application {

   @Override
   public void start(Stage stage) {

         //Definging the axes
         NumberAxis xAxis = new NumberAxis(0, 100, 10);
         xAxis.setLabel("Age");

         NumberAxis yAxis = new NumberAxis(20, 100, 10);
         yAxis.setLabel("Weight");

         //Creating the Bubble chart
         BubbleChart bubbleChart = new BubbleChart(xAxis, yAxis);

         //Prepare XYChart.Series objects by setting data
         XYChart.Series series = new XYChart.Series();
         series.setName("work");

         series.getData().add(new XYChart.Data(10,30,4));
         series.getData().add(new XYChart.Data(25,40,5));
         series.getData().add(new XYChart.Data(40,50,9));
         series.getData().add(new XYChart.Data(55,60,7));
         series.getData().add(new XYChart.Data(70,70,9));
         series.getData().add(new XYChart.Data(85,80,6));
```


```
        //Setting the data to bar chart

        bubbleChart.getData().add(series);

        //Creating a Group object
        Group root = new Group(bubbleChart);

        //Creating a scene object
        Scene scene = new Scene(root, 600, 400);

        //Setting title to the Stage
        stage.setTitle("Bubble Chart");

        //Adding scene to the satge
        stage.setScene(scene);

        //Displaying the contents of the stage
        stage.show();

    }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac BubbleChartExample.java
java BubbleChartExample
```

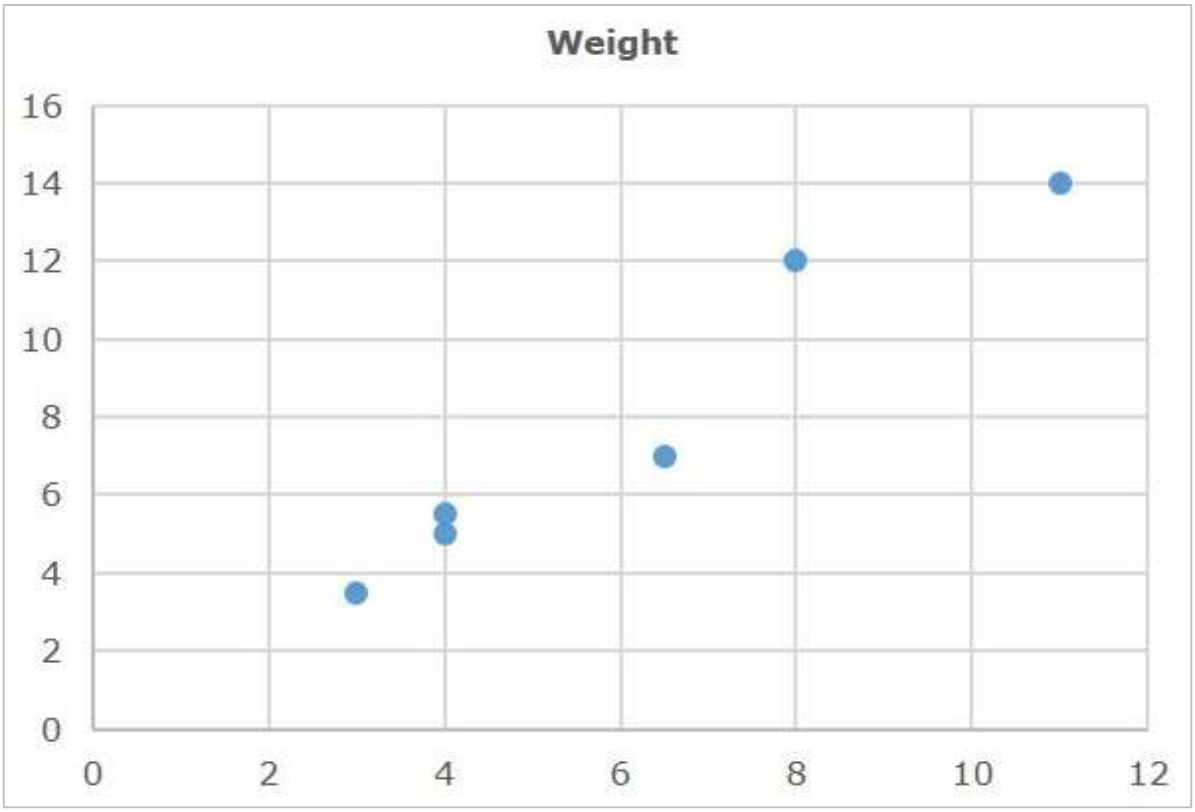
On executing, the above program generates a JavaFX window displaying a bubble chart as shown below.



Charts – Scatter Chart

A scatterplot is a type of graph which uses values from two variables plotted in a Cartesian plane. It is usually used to find out the relationship between two variables.

Following is a Scatter chart plotted between area and weight.



In JavaFX, a Scatter chart is represented by a class named **ScatterChart.** This class belongs to the package **javafx.scene.chart.** By instantiating this class, you can create a ScatterChart node in JavaFX.

To generate an area chart in JavaFX, follow the steps given below.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application**. You can then implement the **start()** method of this class as follows.

```
public class ClassName extends Application {


    @Override
    public void start(Stage primaryStage) throws Exception {


    }


}
```

Step 2: Defining the Axis

Define the X and Y axis of the area chart and set labels to them. In our example, X axis represents area and the Y axis represents weights.

```
//Defining the x axis
NumberAxis xAxis = new NumberAxis(0, 12, 3);
xAxis.setLabel("Area");

//Defining the y axis
NumberAxis yAxis = new NumberAxis(0, 16, 4);
yAxis.setLabel("Weight");
```

Step 3: Creating the Scatter Chart

Create a line chart by instantiating the class named **ScatterChart** of the package **javafx.scene.chart**. To the constructor of this class, pass the objects representing the X and Y axis created in the previous step.

```
//Creating the Scatter chart
ScatterChart<String, Number> scatterChart = new ScatterChart(xAxis, yAxis);
```

Step 4: Preparing the Data

Instantiate the **XYChart.Series** class and add the data (a series of, x and y coordinates) to the Observable list of this class as follows –

```
//Prepare XYChart.Series objects by setting data
XYChart.Series series = new XYChart.Series();
series.getData().add(new XYChart.Data(8, 12));
series.getData().add(new XYChart.Data(4, 5.5));
series.getData().add(new XYChart.Data(11, 14));
series.getData().add(new XYChart.Data(4, 5));
series.getData().add(new XYChart.Data(3, 3.5));
series.getData().add(new XYChart.Data(6.5, 7));
```

Step 5: Add Data to the Scatter Chart

Add the data series prepared in the previous step to the scatter chart as follows –

```
//Setting the data to scatter chart
scatterChart.getData().addAll(series);
```

Step 6: Creating a Group Object

In the **start()** method, create a group object by instantiating the class named **Group**. This belongs to the package **javafx.scene**.

Pass the ScatterChart (node) object created in the previous step as a parameter to the constructor of the Group class. This should be done in order to add it to the group as follows –

```
Group root = new Group(scatterChart);
```

Step 7: Creating a Scene Object

Create a Scene by instantiating the class named **Scene**, which belongs to the package **javafx.scene.** To this class, pass the Group object (**root**) created in the previous step.

In addition to the Root Object, you can also pass two double parameters representing height and width of the screen, along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 8: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object, which is passed to the start method of the scene class as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 9: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using this method as follows.

```
primaryStage.setScene(scene);
```

Step 10: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 11: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows.

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

Example

The following table contains sample data plotted between area and weight.

Area	Weight
8	12
4	5.5
11	14
4	5
3	3.5
6.5	7

Following is a Java program which generates a scatter chart depicting the above data using JavaFX.

Save this code in a file with the name **ScatterChartExample.java.**

```
import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.ScatterChart;
import javafx.scene.chart.XYChart;

public class ScatterChartExample extends Application {

   @Override
   public void start(Stage stage) {

        //Definging the axes
        NumberAxis xAxis = new NumberAxis(0, 12, 3);
```

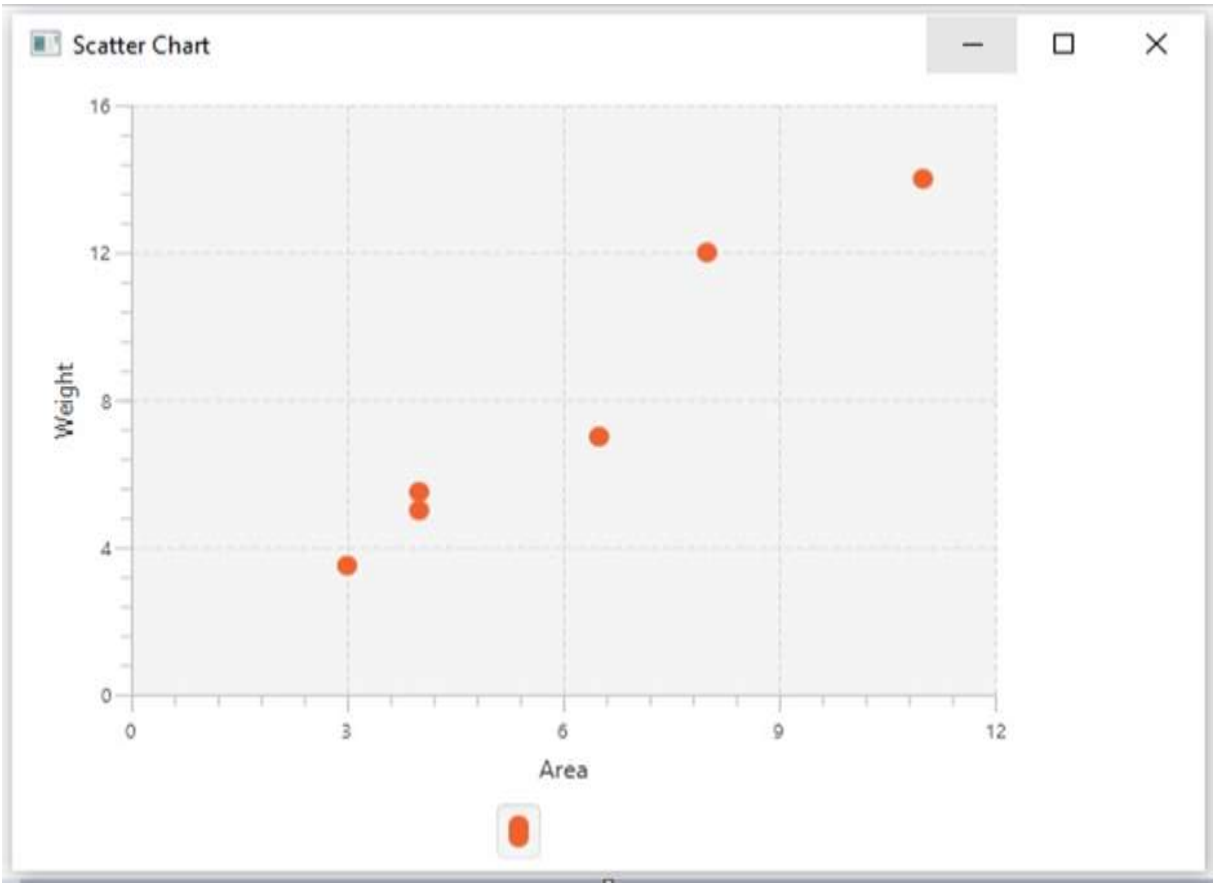
```
        xAxis.setLabel("Area");

        NumberAxis yAxis = new NumberAxis(0, 16, 4);
        yAxis.setLabel("Weight");

        //Creating the Scatter chart
        ScatterChart<String, Number> scatterChart = new ScatterChart(xAxis,
yAxis);

        //Prepare XYChart.Series objects by setting data
        XYChart.Series series = new XYChart.Series();
        series.getData().add(new XYChart.Data(8, 12));
        series.getData().add(new XYChart.Data(4, 5.5));
        series.getData().add(new XYChart.Data(11, 14));
        series.getData().add(new XYChart.Data(4, 5));
        series.getData().add(new XYChart.Data(3, 3.5));
        series.getData().add(new XYChart.Data(6.5, 7));

        //Setting the data to scatter chart
        scatterChart.getData().addAll(series);

        //Creating a Group object
        Group root = new Group(scatterChart);

        //Creating a scene object
        Scene scene = new Scene(root, 600, 400);

        //Setting title to the Stage
        stage.setTitle("Scatter Chart");

        //Adding scene to the satge
        stage.setScene(scene);

        //Displaying the contents of the stage
        stage.show();
```

```
    }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac ScatterChartExample.java
java ScatterChartExample
```

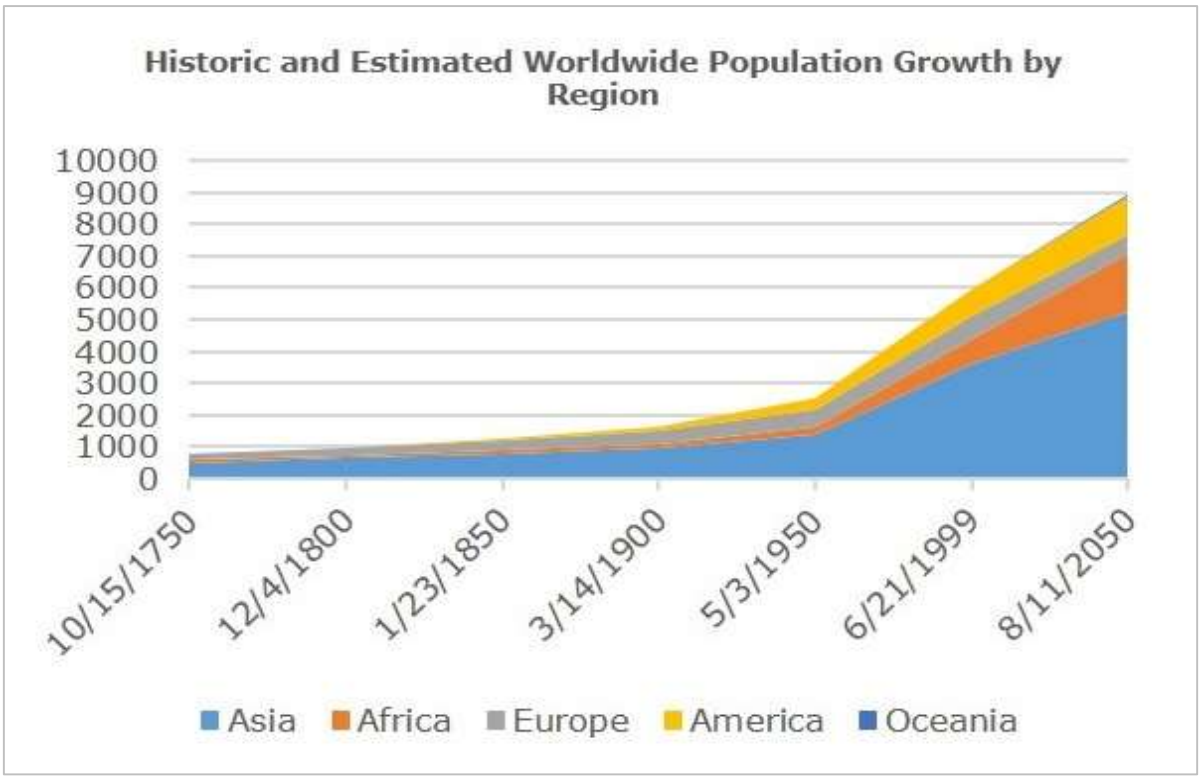
On executing, the above program generates a JavaFX window displaying a scatter chart as shown below.



Charts – StackedArea Chart

StackedArea Chart is a variation of the Area Chart that displays trends of the contribution of each value (For example – overtime). The areas are stacked so that each series adjoins, but does not overlap the preceding series. This contrasts with the Area chart where each series overlays the preceding series.

Following is a Stacked chart depicting population growth.



Source: wikipedia.org

In JavaFX, a Stacked Area chart is represented by a class named **StackedAreaChart.** This class belongs to the package **javafx.scene.chart**. By instantiating this class, you can create a StackedAreaChart node in JavaFX.

To generate a stacked area chart in JavaFX, follow the steps given below.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application**. Then you can implement the **start()** method of this class as follows.

```
public class ClassName extends Application {

    @Override

    public void start(Stage primaryStage) throws Exception {

  }

}
```

Step 2: Defining the Axis

Define the X and Y axis of the stacked area chart and set labels to them. In our example, X axis represents various years from 1750 to 2050. These have major tick units for every 50 years. While the Y axis represents the population growth in millions.

```
//Defining the X axis
CategoryAxis xAxis = new CategoryAxis();

xAxis.setCategories(FXCollections.<String>observableArrayList(Arrays.asList("1
750", "1800", "1850", "1900", "1950", "1999", "2050" )));


Defining the Y axis
NumberAxis yAxis = new NumberAxis(0, 10000, 2500);
yAxis.setLabel("Population in Billions");
```

Step 3: Creating the Stacked Area Chart

Create a line chart by instantiating the class named **StackedAreaChart** of the package **javafx.scene.chart**. To the constructor of this class, pass the objects representing the X and Y axis created in the previous step.

```
//Creating the Area chart
StackedAreaChart<String, Number> areaChart = new StackedAreaChart(xAxis,
yAxis);
areaChart.setTitle("Historic and Estimated Worldwide Population Growth by
Region");
```

Step 4: Preparing the Data

Instantiate the **XYChart.Series** class and add the data (a series of, x and y coordinates) to the Observable list of this class as follows –

```
//Prepare XYChart.Series objects by setting data
XYChart.Series series1 = new XYChart.Series();
series1.setName("Asia");
series1.getData().add(new XYChart.Data("1750", 502));
series1.getData().add(new XYChart.Data("1800", 635));
series1.getData().add(new XYChart.Data("1850", 809));
series1.getData().add(new XYChart.Data("1900", 947));
series1.getData().add(new XYChart.Data("1950", 1402));
series1.getData().add(new XYChart.Data("1999", 3634));
series1.getData().add(new XYChart.Data("2050", 5268));


XYChart.Series series2 = new XYChart.Series();
series2.setName("Africa");
series2.getData().add(new XYChart.Data("1750", 106));
```

```
series2.getData().add(new XYChart.Data("1800", 107));
series2.getData().add(new XYChart.Data("1850", 111));
series2.getData().add(new XYChart.Data("1900", 133));
series2.getData().add(new XYChart.Data("1950", 221));
series2.getData().add(new XYChart.Data("1999", 767));
series2.getData().add(new XYChart.Data("2050", 1766));

XYChart.Series series3 = new XYChart.Series();
series3.setName("Europe");
series3.getData().add(new XYChart.Data("1750", 163));
series3.getData().add(new XYChart.Data("1800", 203));
series3.getData().add(new XYChart.Data("1850", 276));
series3.getData().add(new XYChart.Data("1900", 408));
series3.getData().add(new XYChart.Data("1950", 547));
series3.getData().add(new XYChart.Data("1999", 729));
series3.getData().add(new XYChart.Data("2050", 628));

XYChart.Series series4 = new XYChart.Series();
series4.setName("America");
series4.getData().add(new XYChart.Data("1750", 18));
series4.getData().add(new XYChart.Data("1800", 31));


series4.getData().add(new XYChart.Data("1850", 54));
series4.getData().add(new XYChart.Data("1900", 156));
series4.getData().add(new XYChart.Data("1950", 339));
series4.getData().add(new XYChart.Data("1999", 818));
series4.getData().add(new XYChart.Data("2050", 1201));

XYChart.Series series5 = new XYChart.Series();
series5.setName("Oceania");
series5.getData().add(new XYChart.Data("1750", 2));
series5.getData().add(new XYChart.Data("1800", 2));
series5.getData().add(new XYChart.Data("1850", 2));
series5.getData().add(new XYChart.Data("1900", 6));
series5.getData().add(new XYChart.Data("1950", 13));
```

```
series5.getData().add(new XYChart.Data("1999", 30));
series5.getData().add(new XYChart.Data("2050", 46));
```

Step 5: Add Data to the Stacked Area Chart

Add the data series prepared in the previous step to the stacked area chart as follows –

```
//Setting the data to area chart
areaChart.getData().addAll(series1, series2, series3, series4, series5);
```

Step 6: Creating a Group Object

In the **start()** method, create a group object by instantiating the class named **Group**, which belongs to the package **javafx.scene**.

Pass the StackedAreaChart (node) object created in the previous step as a parameter to the constructor of the Group class. This should be done in order to add it to the group as follows –

```
Group root = new Group(stackedAreaChart);
```

Step 7: Creating a Scene Object

Create a Scene by instantiating the class named **Scene**, which belongs to the package **javafx.scene.** To this class, pass the Group object (**root**) created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen, along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 8: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. The **primaryStage** is a Stage object, which is passed to the start method of the scene class as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 9: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named **Stage.** Add the Scene object prepared in the previous steps using this method as follows.

```
primaryStage.setScene(scene);
```

Step 10: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 11: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows.

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

Example

The following table lists out the population of different continents from year 1750 till year 2050.

	Asia	Africa	Europe	America	Oceania
1750	502	106	163	18	2
1800	635	107	203	31	2
1850	809	111	276	54	2
1900	947	133	408	156	6
1950	1402	221	547	339	13
1999	3634	767	729	818	30
2050	5268	1766	628	1201	46

Following is a Java program which generates a stacked area chart depicting the above data using JavaFX.

Save this code in a file with the name **StackedAreaChartExample.java.**

```
import java.util.Arrays;
import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.collections.FXCollections;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.chart.CategoryAxis;
import javafx.stage.Stage;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.StackedAreaChart;
import javafx.scene.chart.XYChart;

public class StackedAreaChartExample extends Application {

   @Override
   public void start(Stage stage) {

         //Definging the axes
         CategoryAxis xAxis = new CategoryAxis();



            xAxis.setCategories(FXCollections.<String>observableArrayList(
          Arrays.asList("1750", "1800", "1850", "1900", "1950", "1999",
"2050" )));

         NumberAxis yAxis = new NumberAxis(0, 10000, 2500);
         yAxis.setLabel("Population in Millions");

         //Creating the Area chart
         StackedAreaChart<String, Number> areaChart = new
StackedAreaChart(xAxis, yAxis);
         areaChart.setTitle("Historic and Estimated Worldwide Population Growth
by Region");

         //Prepare XYChart.Series objects by setting data
         XYChart.Series series1 = new XYChart.Series();
         series1.setName("Asia");
```

```
        series1.getData().add(new XYChart.Data("1750", 502));
        series1.getData().add(new XYChart.Data("1800", 635));
        series1.getData().add(new XYChart.Data("1850", 809));
        series1.getData().add(new XYChart.Data("1900", 947));
        series1.getData().add(new XYChart.Data("1950", 1402));
        series1.getData().add(new XYChart.Data("1999", 3634));
        series1.getData().add(new XYChart.Data("2050", 5268));

        XYChart.Series series2 = new XYChart.Series();
        series2.setName("Africa");
        series2.getData().add(new XYChart.Data("1750", 106));
        series2.getData().add(new XYChart.Data("1800", 107));
        series2.getData().add(new XYChart.Data("1850", 111));
        series2.getData().add(new XYChart.Data("1900", 133));
        series2.getData().add(new XYChart.Data("1950", 221));
        series2.getData().add(new XYChart.Data("1999", 767));
        series2.getData().add(new XYChart.Data("2050", 1766));

        XYChart.Series series3 = new XYChart.Series();

        series3.setName("Europe");

       series3.getData().add(new XYChart.Data("1750", 163));
        series3.getData().add(new XYChart.Data("1800", 203));
        series3.getData().add(new XYChart.Data("1850", 276));
        series3.getData().add(new XYChart.Data("1900", 408));
        series3.getData().add(new XYChart.Data("1950", 547));
        series3.getData().add(new XYChart.Data("1999", 729));
        series3.getData().add(new XYChart.Data("2050", 628));

        XYChart.Series series4 = new XYChart.Series();
        series4.setName("America");
        series4.getData().add(new XYChart.Data("1750", 18));
        series4.getData().add(new XYChart.Data("1800", 31));
        series4.getData().add(new XYChart.Data("1850", 54));
        series4.getData().add(new XYChart.Data("1900", 156));
```

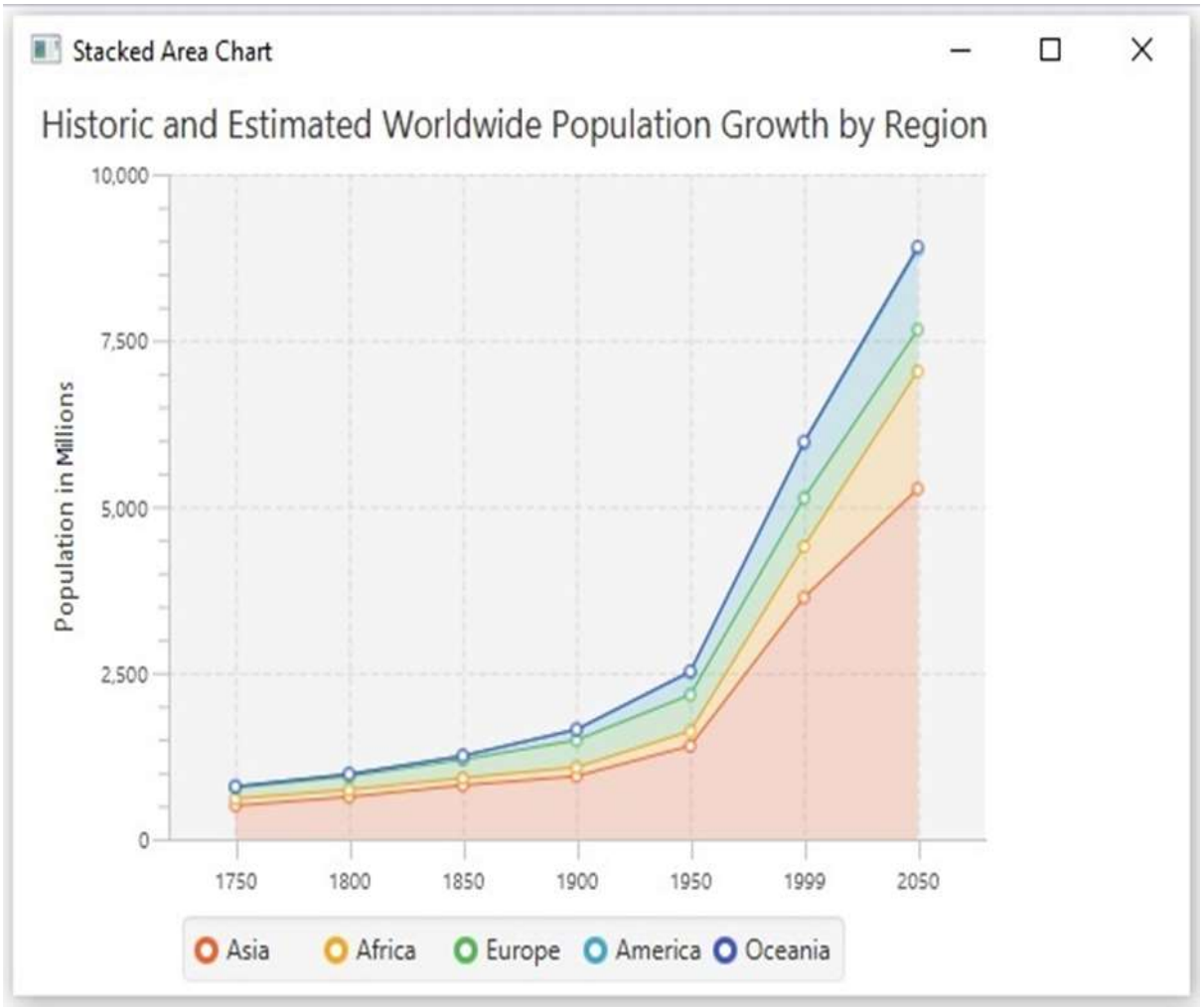
```
        series4.getData().add(new XYChart.Data("1950", 339));
        series4.getData().add(new XYChart.Data("1999", 818));
        series4.getData().add(new XYChart.Data("2050", 1201));

        XYChart.Series series5 = new XYChart.Series();
        series5.setName("Oceania");
        series5.getData().add(new XYChart.Data("1750", 2));
        series5.getData().add(new XYChart.Data("1800", 2));
        series5.getData().add(new XYChart.Data("1850", 2));
        series5.getData().add(new XYChart.Data("1900", 6));
        series5.getData().add(new XYChart.Data("1950", 13));
        series5.getData().add(new XYChart.Data("1999", 30));
        series5.getData().add(new XYChart.Data("2050", 46));

        //Setting the data to area chart
        areaChart.getData().addAll(series1, series2, series3, series4,
series5);

        //Creating a Group object
        Group root = new Group(areaChart);



        //Creating a scene object
        Scene scene = new Scene(root, 600, 400);

        //Setting title to the Stage
        stage.setTitle("Stacked Area Chart");

        //Adding scene to the satge
        stage.setScene(scene);

        //Displaying the contents of the stage
        stage.show();
    }

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

```
        launch(args);
    }
}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac StackedAreaChartExample.java
java StackedAreaChartExample
```

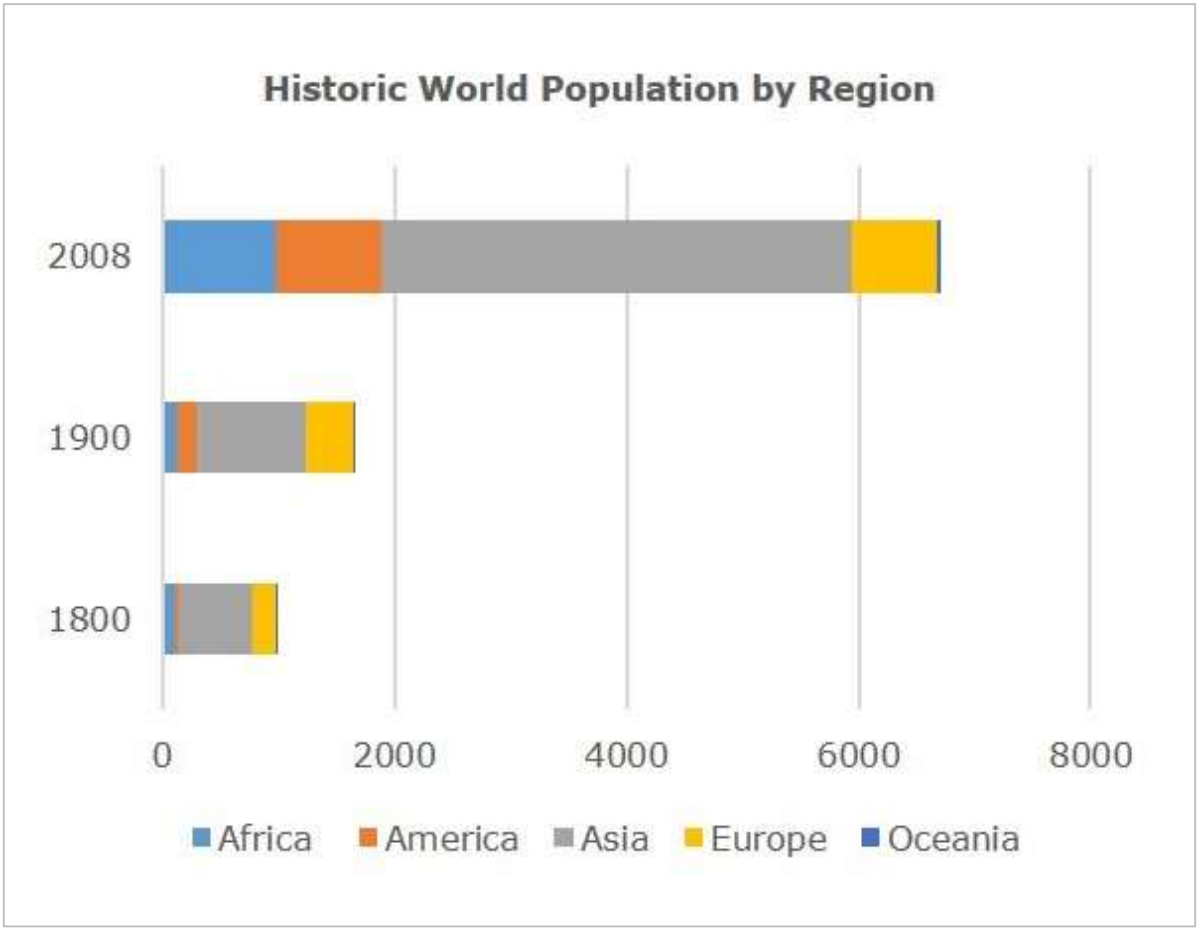
On executing, the above program generates a JavaFX window displaying a stacked area chart as shown below.



Charts – Stacked Bar Chart

StackedBarChart is a variation of a BarChart, which plots bars indicating data values for a category. The bars can be vertical or horizontal depending on which axis is the category axis. The bar for each series is stacked on top of the previous series.

The following is a Stacked Bar Chart, which depicts the population growth.



Source: Wikipedia.org

In JavaFX, a Stacked Bar Chart is represented by a class named **StackedBarChart.** This class belongs to the package **javafx.scene.chart**. By instantiating this class, you can create a StackedBarChart node in JavaFX.

To generate a Stacked Bar Chart in JavaFX, follow the steps given below.

Step 1: Creating a Class

Create a Java class and inherit the **Application** class of the package **javafx.application**. You can then implement the **start()** method of this class as follows.

```
public class ClassName extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
    }
}
```

Step 2: Defining the Axis

Define the X and Y axis of the stacked bar chart and set labels to them. In our example, X axis represents the continents and the y axis represents the population in millions.

```
//Defining the x axis
CategoryAxis xAxis = new CategoryAxis();

xAxis.setCategories(FXCollections.<String>observableArrayList(Arrays.asList("A
frica", "America", "Asia", "Europe", "Oceania")));
xAxis.setLabel("category");


//Defining the y axis
NumberAxis yAxis = new NumberAxis();
yAxis.setLabel("Population (In millions)");
```

Step 3: Creating the Stacked Bar Chart

Create a line chart by instantiating the class named **StackedBarChart** of the package **javafx.scene.chart**. To the constructor of this class, pass the objects representing the X and Y axis created in the previous step.

```
//Creating the Bar chart
StackedBarChart<String, Number> stackedBarChart = new StackedBarChart<>(xAxis,
yAxis);
stackedBarChart.setTitle("Historic World Population by Region");
```

Step 4: Preparing the Data

Instantiate the **XYChart.Series** class and add the data (a series of, x and y coordinates) to the Observable list of this class as follows –

```
//Prepare XYChart.Series objects by setting data
XYChart.Series<String, Number> series1 = new XYChart.Series<>();
series1.setName("1800");
series1.getData().add(new XYChart.Data<>("Africa", 107));
series1.getData().add(new XYChart.Data<>("America", 31));
series1.getData().add(new XYChart.Data<>("Asia", 635));
series1.getData().add(new XYChart.Data<>("Europe", 203));
series1.getData().add(new XYChart.Data<>("Oceania", 2));


XYChart.Series<String, Number> series2 = new XYChart.Series<>();
```

```
series2.setName("1900");
series2.getData().add(new XYChart.Data<>("Africa", 133));
series2.getData().add(new XYChart.Data<>("America", 156));
series2.getData().add(new XYChart.Data<>("Asia", 947));
series2.getData().add(new XYChart.Data<>("Europe", 408));
series1.getData().add(new XYChart.Data<>("Oceania", 6));

XYChart.Series<String, Number> series3 = new XYChart.Series<>();
series3.setName("2008");
series3.getData().add(new XYChart.Data<>("Africa", 973));
series3.getData().add(new XYChart.Data<>("America", 914));
series3.getData().add(new XYChart.Data<>("Asia", 4054));
series3.getData().add(new XYChart.Data<>("Europe", 732));
series1.getData().add(new XYChart.Data<>("Oceania", 34));
```

Step 5: Add Data to the Stacked Bar Chart

Add the data series prepared in the previous step to the bar chart as follows –

```
//Setting the data to bar chart
stackedBarChart.getData().addAll(series1, series2, series3);
```

Step 6: Creating a Group Object

In the **start()** method, create a group object by instantiating the class named **Group**. This belongs to the package **javafx.scene**.

Pass the StackedBarChart (node) object created in the previous step as a parameter to the constructor of the Group class. This should be done in order to add it to the group as follows –

```
Group root = new Group(stackedBarChart);
```

Step 7: Creating a Scene Object

Create a Scene by instantiating the class named Scene, which belongs to the package **javafx.scene**. To this class, pass the Group object (root), created in the previous step.

In addition to the root object, you can also pass two double parameters representing height and width of the screen, along with the object of the Group class as follows.

```
Scene scene = new Scene(group ,600, 300);
```

Step 8: Setting the Title of the Stage

You can set the title to the stage using the **setTitle()** method of the **Stage** class. This **primaryStage** is a Stage object, which is passed to the start method of the scene class as a parameter.

Using the **primaryStage** object, set the title of the scene as **Sample Application** as follows.

```
primaryStage.setTitle("Sample Application");
```

Step 9: Adding Scene to the Stage

You can add a Scene object to the stage using the method **setScene()** of the class named Stage. Add the Scene object prepared in the previous steps using this method as follows.

```
primaryStage.setScene(scene);
```

Step 10: Displaying the Contents of the Stage

Display the contents of the scene using the method named **show()** of the **Stage** class as follows.

```
primaryStage.show();
```

Step 11: Launching the Application

Launch the JavaFX application by calling the static method **launch()** of the **Application** class from the main method as follows.

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

Example

The following table lists out the population in various continents in the years 1800, 1900 and 2008.

	Africa	America	Asia	Europe	Oceania

1800	107	31	635	203	2
1900	133	156	947	408	6
2008	973	914	4054	732	34

Following is a Java program that generates a stacked bar chart depicting the above data, using JavaFX.

Save this code in a file with the name **StackedBarChartExample.java.**

```
import java.util.Arrays;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.chart.CategoryAxis;
import javafx.stage.Stage;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.StackedBarChart;
import javafx.scene.chart.XYChart;

public class StackedBarChartExample extends Application {

   @Override
   public void start(Stage stage) {

         //Definging the axes
         CategoryAxis xAxis = new CategoryAxis();

xAxis.setCategories(FXCollections.<String>observableArrayList(Arrays.asList("A
frica", "America", "Asia", "Europe", "Oceania")));
         xAxis.setLabel("category");

         NumberAxis yAxis = new NumberAxis();
         yAxis.setLabel("Population (In millions)");
```

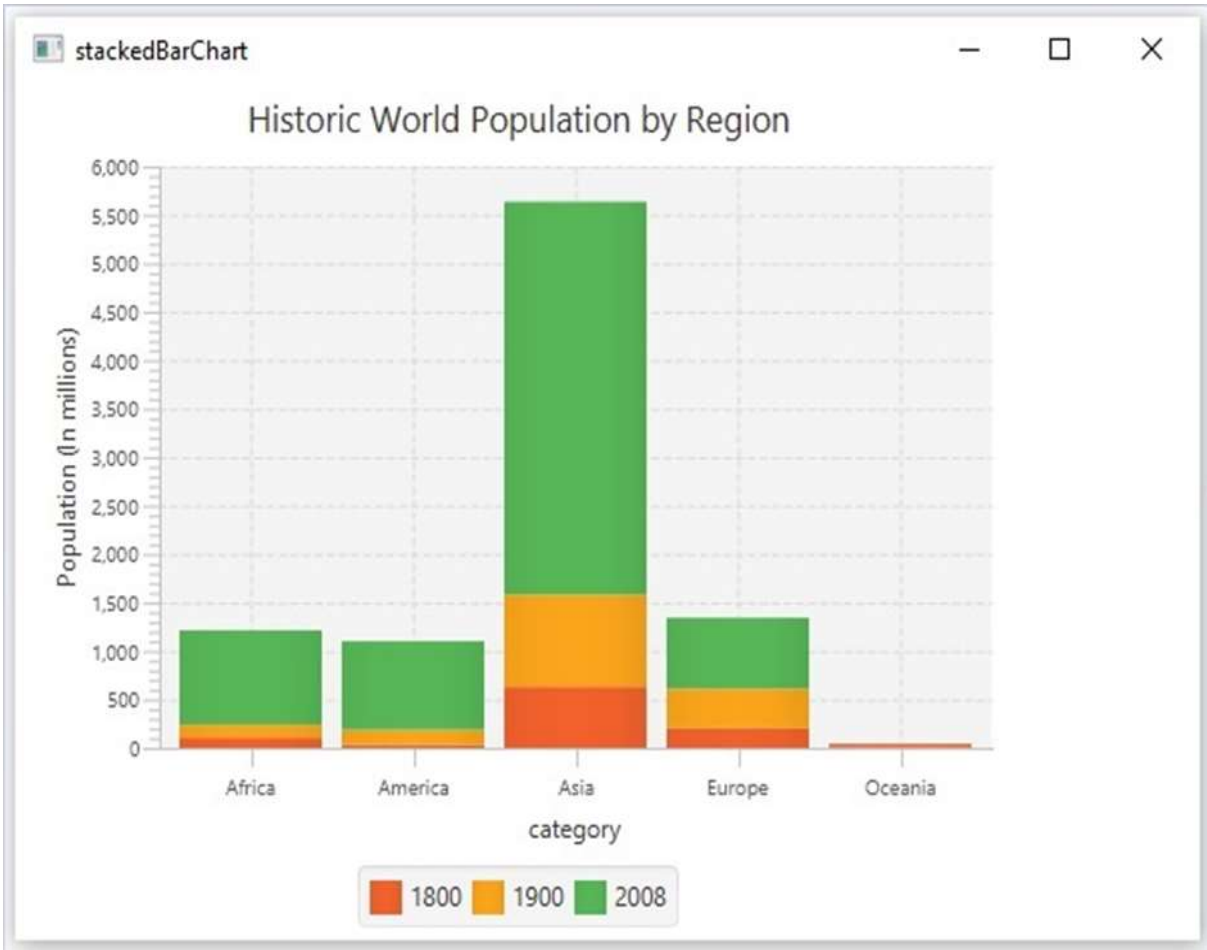
```
        //Creating the Bar chart
        StackedBarChart<String, Number> stackedBarChart = new
StackedBarChart<>(xAxis, yAxis);
        stackedBarChart.setTitle("Historic World Population by Region");

        //Prepare XYChart.Series objects by setting data
        XYChart.Series<String, Number> series1 = new XYChart.Series<>();

        series1.setName("1800");
        series1.getData().add(new XYChart.Data<>("Africa", 107));
        series1.getData().add(new XYChart.Data<>("America", 31));

        series1.getData().add(new XYChart.Data<>("Asia", 635));
        series1.getData().add(new XYChart.Data<>("Europe", 203));
        series1.getData().add(new XYChart.Data<>("Oceania", 2));

        XYChart.Series<String, Number> series2 = new XYChart.Series<>();
        series2.setName("1900");
        series2.getData().add(new XYChart.Data<>("Africa", 133));
        series2.getData().add(new XYChart.Data<>("America", 156));
        series2.getData().add(new XYChart.Data<>("Asia", 947));
        series2.getData().add(new XYChart.Data<>("Europe", 408));
        series1.getData().add(new XYChart.Data<>("Oceania", 6));

        XYChart.Series<String, Number> series3 = new XYChart.Series<>();
        series3.setName("2008");
        series3.getData().add(new XYChart.Data<>("Africa", 973));
        series3.getData().add(new XYChart.Data<>("America", 914));
        series3.getData().add(new XYChart.Data<>("Asia", 4054));
        series3.getData().add(new XYChart.Data<>("Europe", 732));
        series1.getData().add(new XYChart.Data<>("Oceania", 34));

        //Setting the data to bar chart
        stackedBarChart.getData().addAll(series1, series2, series3);

        //Creating a Group object
```

```
        Group root = new Group(stackedBarChart);

        //Creating a scene object
        Scene scene = new Scene(root, 600, 400);

        //Setting title to the Stage
        stage.setTitle("stackedBarChart");

        //Adding scene to the satge
        stage.setScene(scene);

        //Displaying the contents of the stage
        stage.show();
   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac StackedBarChartExample.java
java StackedBarChartExample
```

On executing, the above program generates a JavaFX window displaying an area chart as shown below.



16. JavaFX – Layout Panes (Containers)

After constructing all the required nodes in a scene, we will generally arrange them in order.

This arrangement of the components within the container is called the Layout of the container. We can also say that we followed a layout as it includes placing all the components at a particular position within the container.

JavaFX provides several predefined layouts such as **HBox**, **VBox**, **Border Pane**, **Stack Pane**, **Text Flow**, **Anchor Pane**, **Title Pane**, **Grid Pane**, **Flow Pane**l, etc.

Each of the above mentioned layout is represented by a class and all these classes belongsto the package **javafx.layout**. The class named **Pane** is the base class of all the layouts in JavaFX.

Creating a Layout

To create a layout, you need to –

- Create nodes.
- Instantiate the respective class of the required layout.
- Set the properties of the layout.
- Add all the created nodes to the layout.

Creating Nodes

First of all, create the required nodes of the JavaFX application by instantiating their respective classes.

For example, if you want to have a text field and two buttons namely, play and stop in a HBox layout - you will have to initially create those nodes as shown in the following code block:

```
//Creating a text field
TextField textField = new TextField();
//Creating the play button
Button playButton = new Button("Play");
//Creating the stop button
Button stopButton = new Button("stop");
```

Instantiating the Respective Class

After creating the nodes (and completing all the operations on them), instantiate the class of the required layout.

For Example, if you want to create a Hbox layout, you need to instantiate this class as follows.

```
HBox hbox = new HBox();
```

Setting the Properties of the Layout

After instantiating the class, you need to set the properties of the layout using their respective setter methods.

For example: If you want to set space between the created nodes in the HBox layout, then you need to set value to the property named spacing. This can be done by using the setter method **setSpacing()** as shown below –

```
hbox.setSpacing(10);
```

Adding the Shape Object to the Group

Finally, you need to add the object of the shape to the group by passing it as a parameter of the constructor as shown below.

```
//Creating a Group object
Group root = new Group(line);
```

Layout Panes

Following are the various Layout panes (classes) provided by JavaFX. These classes exist in the package **javafx.scene.layout.**

S.No	Shape and Description
1	**HBox:** The HBox layout arranges all the nodes in our application in a single horizontal row. The class named **HBox** of the package **javafx.scene.layout** represents the text horizontal box layout.
2	**VBox:** The VBox layout arranges all the nodes in our application in a single vertical column. The class named **VBox** of the package **javafx.scene.layout** represents the text Vertical box layout.

3	**BorderPane:** The Border Pane layout arranges the nodes in our application in top, left, right, bottom and center positions. The class named **BorderPane** of the package **javafx.scene.layout** represents the border pane layout.
4	**StackPane:** The stack pane layout arranges the nodes in our application on top of another just like in a stack. The node added first is placed at the bottom of the stack and the next node is placed on top of it. The class named **StackPane** of the package **javafx.scene.layout** represents the stack pane layout.
5	**TextFlow:** The Text Flow layout arranges multiple text nodes in a single flow. The class named **TextFlow** of the package **javafx.scene.layout** represents the text flow layout.
6	**AnchorPane:** The Anchor pane layout anchors the nodes in our application at a particular distance from the pane. The class named **AnchorPane** of the package **javafx.scene.layout** represents the Anchor Pane layout.
7	**TilePane:** The Tile Pane layout adds all the nodes of our application in the form of uniformly sized tiles. The class named **TilePane** of the package **javafx.scene.layout** represents the TilePane layout.
8	**GridPane:** The Grid Pane layout arranges the nodes in our application as a grid of rows and columns. This layout comes handy while creating forms using JavaFX. The class named **GridPane** of the package **javafx.scene.layout** represents the GridPane layout.

9	**FlowPane:** The flow pane layout wraps all the nodes in a flow. A horizontal flow pane wraps the elements of the pane at its height, while a vertical flow pane wraps the elements at its width. The class named **FlowPane** of the package **javafx.scene.layout** represents the Flow Pane layout.

HBox

If we use HBox in the layout in our application, all the nodes are set in a single horizontal row.

The class named **HBox** of the package **javafx.scene.layout** represents the HBox pane. This class contains five properties namely –

- **alignment:** This property represents the alignment of the nodes in the bounds of the HBox. You can set value to this property using the setter method **setAlignment().**

- **fillHeight**: This property is of Boolean type and on setting this to true, the resizable nodes in the HBox are resized to the height of the HBox. You can set value to this property using the setter method **setFillHeight().**

- **spacing**: This property is of double type and it represents the space between the children of the HBox. You can set value to this property using the setter method **setSpacing().**

In addition to these, this class also provides a couple of methods, which are –

- **setHgrow():** Sets the horizontal grow priority for the child when contained by an HBox. This method accepts a node and a priority value.

- **setMargin():** Using this method, you can set margins to the HBox. This method accepts a node and an object of the Insets class (A set of inside offsets for the 4 side of a rectangular area).

Example

The following program is an example of the HBox layout. Here, we are inserting a text field and two buttons, play and stop. This is done with a spacing of 10 and each having margins with dimensions – (10, 10, 10, 10).

Save this code in a file with the name **HBoxExample.java.**

```
import javafx.application.Application;
import javafx.collections.ObservableList;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Button;
```

```
import javafx.scene.control.TextField;
import javafx.stage.Stage;
import javafx.scene.layout.HBox;

public class HBoxExample extends Application {
   @Override
   public void start(Stage stage) {

      //creating a text field


      TextField textField = new TextField();
      //Creating the play button
      Button playButton = new Button("Play");
      //Creating the stop button
      Button stopButton = new Button("stop");

      //Instantiating the HBox class
      HBox hbox = new HBox();

      //Setting the space between the nodes of a HBox pane
      hbox.setSpacing(10);

      //Setting the margin to the nodes
      hbox.setMargin(textField, new Insets(20, 20, 20, 20));
      hbox.setMargin(playButton, new Insets(20, 20, 20, 20));
      hbox.setMargin(stopButton, new Insets(20, 20, 20, 20));

      //retrieving the observable list of the HBox
      ObservableList list = hbox.getChildren();

      //Adding all the nodes to the observable list (HBox)
      list.addAll(textField, playButton, stopButton);

      //Creating a scene object
      Scene scene = new Scene(hbox);
```

```
      //Setting title to the Stage
      stage.setTitle("Hbox Example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();

   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac HBoxExample.java
java HBoxExample.java
```

On executing, the above program generates a JavaFX window as shown below.

VBox

If we use VBox as the layout in our application, all the nodes are set in a single vertical column.

The class named **VBox** of the package **javafx.scene.layout** represents the VBox pane. This class contains five properties, which are –

- **alignment:** This property represents the alignment of the nodes inside the bounds of the VBox. You can set value to this property by using the setter method **setAlignment().**

- **fillHeight**: This property is of Boolean type and on setting this to be true; the resizable nodes in the VBox are resized to the height of the HBox. You can set value to this property using the setter method **setFillHeight().**

- **spacing**: This property is of double type and it represents the space between the children of the VBox. You can set value to this property using the setter method **setSpacing().**

In addition to these, this class also provides the following methods:

- **setHgrow():** Sets the horizontal grow priority for the child when contained by a VBox. This method accepts a node and a priority value.

- **setMargin():** Using this method, you can set margins to the VBox. This method accepts a node and an object of the Insets class (A set of inside offsets for the 4 sides of a rectangular area)

Example

The following program is an example of the **VBox** layout. In this, we are inserting a text field and two buttons, play and stop. This is done with a spacing of 10 and each having margins with dimensions – (10, 10, 10, 10).

Save this code in a file with the name **VBoxExample.java.**

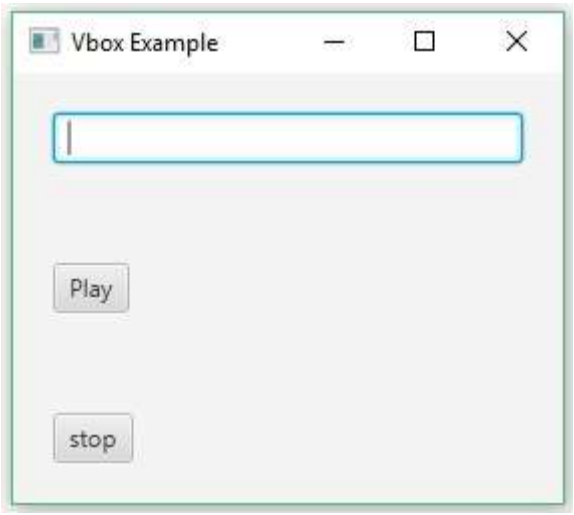
```
import javafx.application.Application;
import javafx.collections.ObservableList;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.TextField;
import javafx.stage.Stage;
import javafx.scene.layout.VBox;

public class VBoxExample extends Application {
   @Override
   public void start(Stage stage) {

      //creating a text field
      TextField textField = new TextField();
      //Creating the play button
      Button playButton = new Button("Play");
      //Creating the stop button
      Button stopButton = new Button("stop");
```

```
      //Instantiating the HBox class
      VBox vBox = new VBox();
      //Setting the space between the nodes of a HBox pane
      vBox.setSpacing(10);
      //Setting the margin to the nodes
      vBox.setMargin(textField, new Insets(20, 20, 20, 20));

      vBox.setMargin(playButton, new Insets(20, 20, 20, 20));
      vBox.setMargin(stopButton, new Insets(20, 20, 20, 20));

      //retrieving the observable list of the HBox
      ObservableList list = vBox.getChildren();
      //Adding all the nodes to the observable list
      list.addAll(textField, playButton, stopButton);

      //Creating a scene object
      Scene scene = new Scene(vBox);

      //Setting title to the Stage
      stage.setTitle("Hbox Example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac VBoxExample.java
java VBoxExample.java
```

On executing, the above program generates a JavaFX window as shown below.



BorderPane

If we use the BorderPane, the nodes are arranged in the Top, Left, Right, Bottom and Center positions.

The class named **BorderPane** of the package **javafx.scene.layout** represents the BorderPane.

This class contains five properties, which include:

- **bottom:** This property is of **Node** type and it represents the node placed at the bottom of the BorderPane. You can set value to this property using the setter method **setBottom().**

- **center**: This property is of **Node** type and it represents the node placed at the center of the BorderPane. You can set value to this property using the setter method **setCenter().**

- **left**: This property is of **Node** type and it represents the node placed at the left of the BorderPane. You can set value to this property using the setter method **setLeft().**

- **right:** This property is of **Node** type and it represents the node placed at the right of the BorderPane. You can set value to this property using the setter method **setRight().**

- **top:** This property is of **Node** type and it represents the node placed at the top of the BorderPane. You can set value to this property using the setter method **setTop().**

In addition to these, this class also provides the following method:

- **setAlignment():** This method is used to set the alignment of the nodes belonging to this pane. This method accepts a node and a priority value.

Example

The following program is an example of the **BorderPane** layout. In this, we are inserting a five text fields in the Top, Bottom, Right, Left and Center positions.

Save this code in a file with the name **BorderPaneExample.java.**

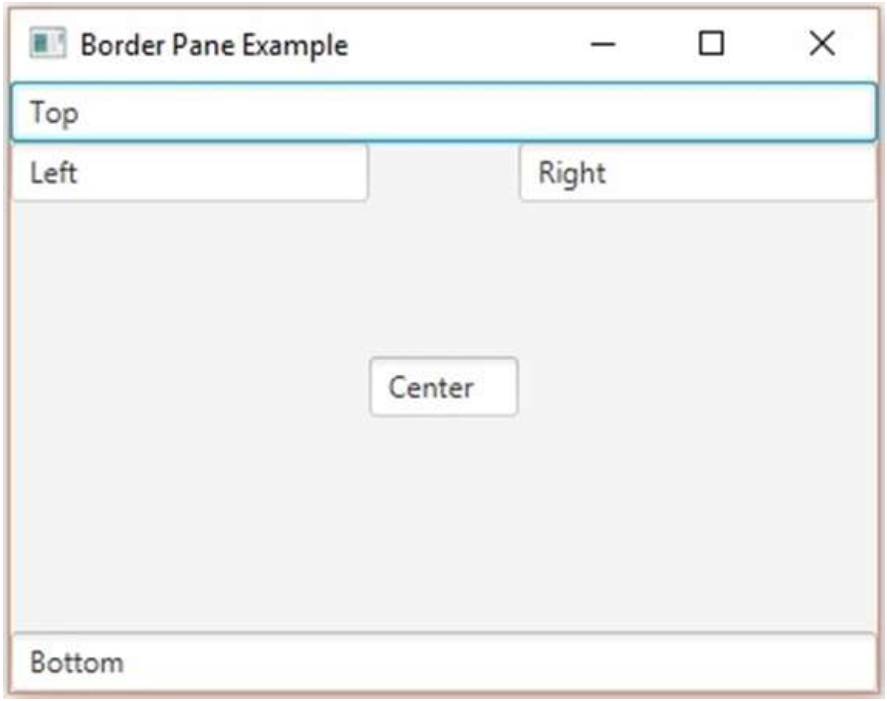
```
import javafx.application.Application;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class BorderPaneExample extends Application {

   @Override
   public void start(Stage stage) {

      //Instantiating the HBox class
      BorderPane bPane = new BorderPane();

      //Setting the top, bottom, center, right and left nodes to the pane
      bPane.setTop(new TextField("Top"));
      bPane.setBottom(new TextField("Bottom"));
      bPane.setLeft(new TextField("Left"));
      bPane.setRight(new TextField("Right"));
      bPane.setCenter(new TextField("Center"));
      //Creating a scene object
      Scene scene = new Scene(bPane);

      //Setting title to the Stage
      stage.setTitle("Hbox Example");

      //Adding scene to the satge
      stage.setScene(scene);
```

```
      //Displaying the contents of the stage
      stage.show();
   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac BorderPaneExample.java
java BorderPaneExample
```

On executing, the above program generates a JavaFX window as shown below.



StackPane

If we use the Stack Pane, the nodes are arranged on top of another, just like in stack. The node added first is placed at the bottom of the stack and the next node is placed on top of it.

The class named **StackPane** of the package **javafx.scene.layout** represents the StackPane. This class contains a single property named alignment. This property represents the alignment of the nodes within the sack pane.

In addition to these, this class also provides a method named **setMargin().** This method is used to set margin for the node within the stack pane.

Example

The following program is an example of the **StackPane** layout. In this, we are inserting a Circle, Sphere and a Text in the same order.

Save this code in a file with the name **StackPaneExample.java.**

```
import javafx.application.Application;
import javafx.collections.ObservableList;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Sphere;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class StackPaneExample extends Application {

   @Override
   public void start(Stage stage) {
      //Drawing a Circle
      Circle circle = new Circle(300, 135, 100);
      circle.setFill(Color.DARKSLATEBLUE);
      circle.setStroke(Color.BLACK);
```

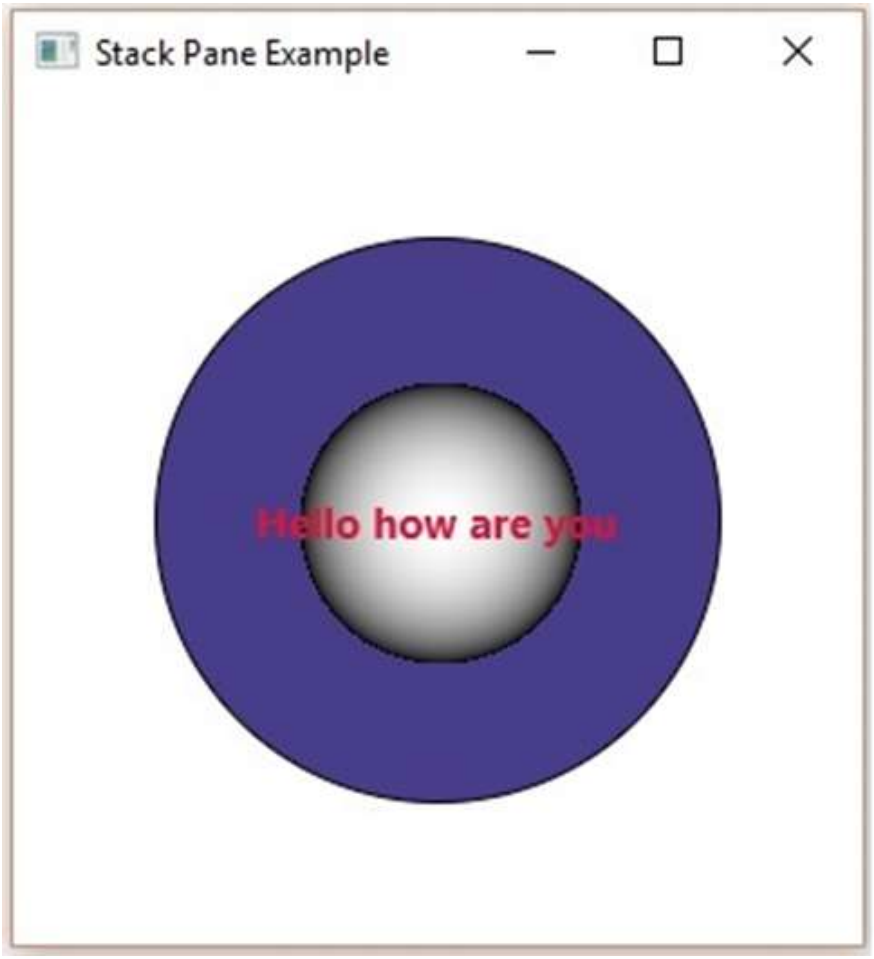
```
      //Drawing Sphere
      Sphere sphere = new Sphere(50);

      //creating a text
      Text text = new Text("Hello how are you");
      //Setting the font of the text
```

```
      text.setFont(Font.font(null, FontWeight.BOLD, 15));
      //Setting the color of the text
      text.setFill(Color.CRIMSON);
      //setting the position of the text
      text.setX(20);
      text.setY(50);

      //Creating a Stackpane
      StackPane stackPane = new StackPane();
      //Setting the margin for the circle
      stackPane.setMargin(circle, new Insets(50, 50, 50, 50));

      //Retrieving the observable list of the Stack Pane
      ObservableList list = stackPane.getChildren();
      //Adding all the nodes to the pane
      list.addAll(circle, sphere, text);

      //Creating a scene object
      Scene scene = new Scene(stackPane);

      //Setting title to the Stage
      stage.setTitle("Stack Pane Example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac StackPaneExample.java
java StackPaneExample
```

On executing, the above program generates a JavaFX window as shown below.



FlowPane

If we use flow pane in our application, all the nodes are wrapped in a flow. A horizontal flow pane wraps the elements of the pane at its height, while a vertical flow pane wraps the elements at its width.

The class named **FlowPane** of the package **javafx.scene.layout** represents the Flow Pane. This class contains 7 properties, which includes –

- **alignment:** This property represents the alignment of the contents of the Flow pane. You can set this property using the setter method **setAllignment().**
- **columnHalignment:** This property represents the horizontal alignments of nodes in a vertical flow pane.
- **rowValignment:** This property represents the vertical alignment of nodes in a horizontal flow pane.

- **Hgap:** This property is of double type and it represents the horizontal gap between the rows/columns of a flow pane.
- **Orientation:** This property represents the orientation of a flow pane.
- **Vgap:** This property is of double type and it represents the vertical gap between the rows/columns of a flow pane.

Example

The following program is an example of the **FlowPane** layout. In this, we are inserting four button in the horizontal flow pane.

Save this code in a file with the name **FlowPaneExample.java.**

```
import javafx.collections.ObservableList;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.FlowPane;
import javafx.scene.shape.Sphere;
import javafx.stage.Stage;

public class FlowPaneExample extends Application {

   @Override
   public void start(Stage stage) {

      //Creating button1
      Button button1 = new Button("Button1");
      //Creating button2
      Button button2 = new Button("Button2");
      //Creating button3
```

```
      Button button3 = new Button("Button3");
      //Creating button4
      Button button4 = new Button("Button4");

      //Creating a Flow Pane
      FlowPane flowPane = new FlowPane();
```

```
      //Setting the horizontal gap between the nodes
      flowPane.setHgap(25);

      //Setting the margin of the pane
     flowPane.setMargin(button1, new Insets(20, 0, 20, 20));

      //Retrieving the observable list of the flow Pane
      ObservableList list = flowPane.getChildren();

      //Adding all the nodes to the flow pane
      list.addAll(button1, button2, button3, button4);

      //Creating a scene object
      Scene scene = new Scene(flowPane);

      //Setting title to the Stage
      stage.setTitle("Flow Pane Example");

      //Adding scene to the stage
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }

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

```
      launch(args);
   }
}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac FlowPaneExample.java
```

```
java FlowPaneExample
```

On executing, the above program generates a JavaFX window as shown below.

AnchorPane

The Anchor Pane 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.

If we use an Anchor pane in our application, the nodes in it are anchored at a particular distance from the pane.

The class named **AnchorPane** of the package **javafx.scene.layout** represents the Anchor Pane. After a node is added, you need to set an anchor to it from the bounds of the pane in all directions (Top, Bottom, Right and Left). To set the anchor, this class provides four methods, which are – **setBottomAnchor()**, **setTopAnchor()**, **setLeftAnchor()**, **setRightAnchor()**. To these methods, you need to pass a double value representing the anchor.

Example

The following program is an example of the Anchor Pane layout. In this, we are inserting a rotating cylinder in an anchor pane. At the same time, we are setting it at a distance of 50 units from the pane from all directions (Top, Left, Right, Bottom).

Save this code in a file with the name **AnchorPaneExample.java.**

```
import javafx.animation.RotateTransition;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.layout.AnchorPane;
import javafx.scene.paint.Color;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.shape.Cylinder;
import javafx.scene.transform.Rotate;
import javafx.stage.Stage;
import javafx.util.Duration;
```

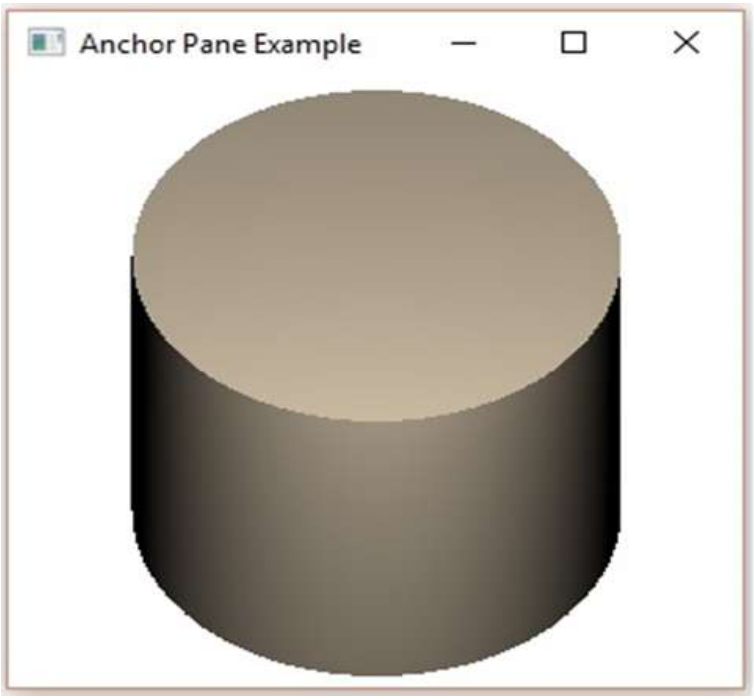
```
public class AnchorPaneExample extends Application {

   @Override
   public void start(Stage stage) {

      //Drawing a Cylinder
      Cylinder cylinder = new Cylinder();
      //Setting the properties of the Cylinder
      cylinder.setHeight(180.0f);
      cylinder.setRadius(100.0f);

      //Preparing the phong material of type diffuse color
      PhongMaterial material = new PhongMaterial();
      material.setDiffuseColor(Color.BLANCHEDALMOND);
      //Setting the diffuse color material to Cylinder5
      cylinder.setMaterial(material);

      //Setting rotation transition for the cylinder
      RotateTransition rotateTransition = new RotateTransition();
      //Setting the duration for the transition
      rotateTransition.setDuration(Duration.millis(1000));
      //Setting the node for the transition
      rotateTransition.setNode(cylinder);
      //Setting the axis of the rotation
      rotateTransition.setAxis(Rotate.X_AXIS);

      //Setting the angle of the rotation
      rotateTransition.setByAngle(360);
      //Setting the cycle count for the transition
      rotateTransition.setCycleCount(RotateTransition.INDEFINITE);
      //Setting auto reverse value to false
      rotateTransition.setAutoReverse(false);
      //playing the animation
      rotateTransition.play();
```

```
      //Creating an Anchor Pane

      AnchorPane anchorPane = new AnchorPane();

      //Setting the anchor to the cylinder
      AnchorPane.setTopAnchor(cylinder, 50.0);
      AnchorPane.setLeftAnchor(cylinder, 50.0);
      AnchorPane.setRightAnchor(cylinder, 50.0);
      AnchorPane.setBottomAnchor(cylinder, 50.0);

      //Retrieving the observable list of the Anchor Pane
      ObservableList list = anchorPane.getChildren();

      //Adding cylinder to the pane
      list.addAll(cylinder);

      //Creating a scene object
      Scene scene = new Scene(anchorPane);

      //Setting title to the Stage
      stage.setTitle("Anchor Pane Example");

      //Adding scene to the satge
      stage.setScene(scene);


      //Displaying the contents of the stage
      stage.show();
   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac AnchorPaneExample.java
java AnchorPaneExample
```

On executing, the above program generates a JavaFX window as shown below.



TextFlow

If we use this layout, you can set multiple text nodes in a single flow. The class named **textFlow** of the package **javafx.scene.layout** represents the text flow.

This class provides two properties, which are –

- **lineSpacing**: This property is of double type and it is used to define the space between the text objects. You can set this property using the method named **setLineSpacing().**

- **textAlignment:** this property represents the alignment of the text objects in the pane. You can set value to this property using the method **setTextAlignment().** To this method you can pass four values: CENTER, JUSTIFY, LEFT, RIGHT.

Example

The following program is an example of the text flow layout. In this, we are creating three text objects with font 15 and with various colors. We are then adding them to a Text flow pane with an alignment value – **Justify**, while the line spacing is **15**.

Save this code in a file with the name **TextFlowExample.java.**

```
import javafx.application.Application;
import javafx.collections.ObservableList;
```

```
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.scene.text.TextAlignment;
import javafx.scene.text.TextFlow;
import javafx.stage.Stage;

public class TextFlowExample extends Application {

   @Override
   public void start(Stage stage) {

      //Creating text objects
      Text text1 = new Text("Welcome to Tutorialspoint ");
      //Setting font to the text
      text1.setFont(new Font(15));
      //Setting color to the text

      text1.setFill(Color.DARKSLATEBLUE);

      Text text2 = new Text("We provide free tutorials for readers in various
technologies  ");
      //Setting font to the text
      text2.setFont(new Font(15));
      //Setting color to the text
      text2.setFill(Color.DARKGOLDENROD);

      Text text3 = new Text("\n Recently we started free video tutorials too
");
      //Setting font to the text
      text3.setFont(new Font(15));
      //Setting color to the text
      text3.setFill(Color.DARKGRAY);

      Text text4 = new Text("We believe in easy learning");
      //Setting font to the text
```

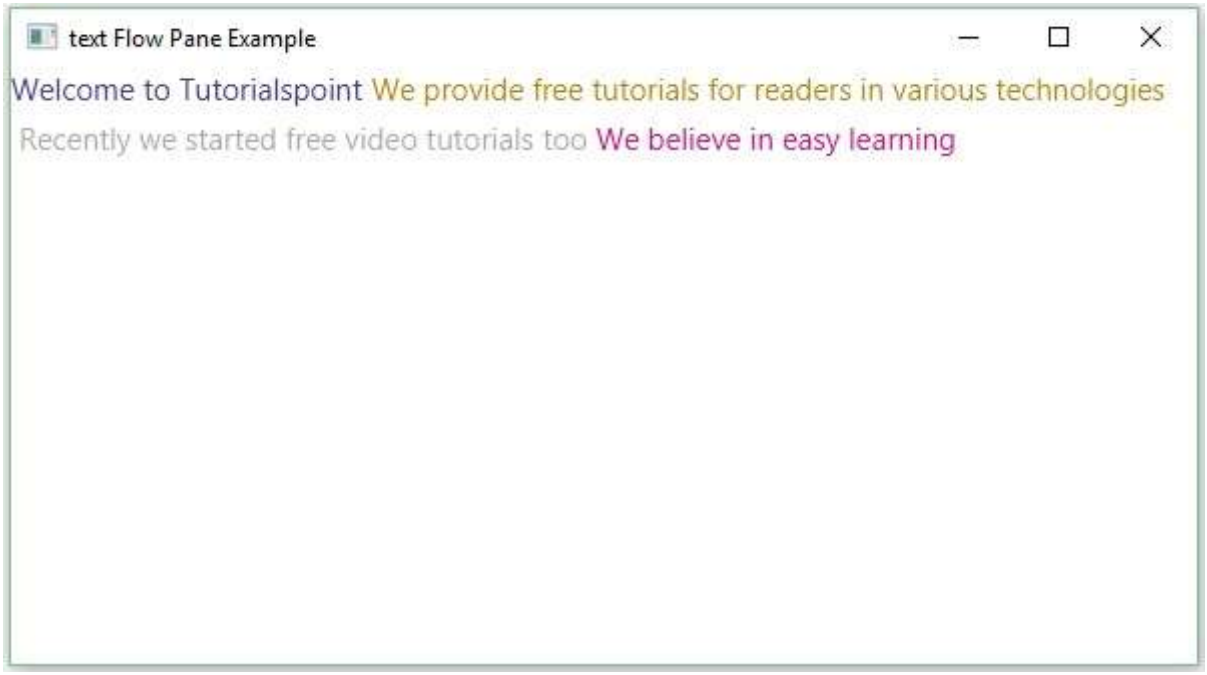
```
      text4.setFont(new Font(15));
      text4.setFill(Color.MEDIUMVIOLETRED);

      //Creating the text flow plane
      TextFlow textFlowPane = new TextFlow();

      //Setting the line spacing between the text objects
      textFlowPane.setTextAlignment(TextAlignment.JUSTIFY);

      //Setting the width
      textFlowPane.setPrefSize(600, 300);

      //Setting the line spacing
      textFlowPane.setLineSpacing(5.0);

      //Retrieving the observable list of the Anchor Pane
      ObservableList list = textFlowPane.getChildren();
      //Adding cylinder to the pane

      list.addAll(text1, text2, text3, text4);

      //Creating a scene object
      Scene scene = new Scene(textFlowPane);

      //Setting title to the Stage
      stage.setTitle("text Flow Pane Example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }

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

```
    }
}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac TextFlowExample.java
java TextFlowExample
```

On executing, the above program generates a JavaFX window as shown below.



TilePane

If we use this pane in our application, all the nodes added to it are arranged in the form of uniformly sized tiles. The class named **tileFlow** of the package **javafx.scene.layout** represents the TilePane flow.

This class provides eleven properties, which are –

- **alignment:** This property represents the alignment of the pane and you can set the value of this property using the **setAlignment()** method.
- **hgap:** This property is of the type double and it represents the horizontal gap between each tile in a row.
- **vgap:** This property is of the type double and it represents the vertical gap between each tile in a row.
- **orientation:** This property represents the orientation of tiles in a row.

- **prefColumns:** This property is of double type and it represents the preferred number of columns for a horizontal tile pane.
- **prefRows:** This property is of double type and it represents the preferred number of rows for a vertical tile pane.
- **prefTileHeight:** This property is of double type and it represents the preferred height of each tile.
- **prefTileWidth:** This property is of double type and it represents the preferred width of each tile.
- **tileHeight:** This property is of double type and it represents the actual height of each tile.
- **tileWidth:** This property is of double type and it represents the actual width of each tile.
- **tileAlignment:** This property is of double type and it represents the default alignment of each child within its tile.

Example

The following program is an example of the tile pane layout. In this, we are creating a tile pane which holds 7 buttons.

Save this code in a file with the name **TilePaneExample.java.**

```
import javafx.application.Application;
import javafx.collections.ObservableList;
import javafx.geometry.Orientation;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.TilePane;
import javafx.stage.Stage;

public class TilePaneExample extends Application {

   @Override
   public void start(Stage stage) {

      //Creating an array of Buttons
      Button[] buttons = new Button[] {
         new Button("SunDay"),
```

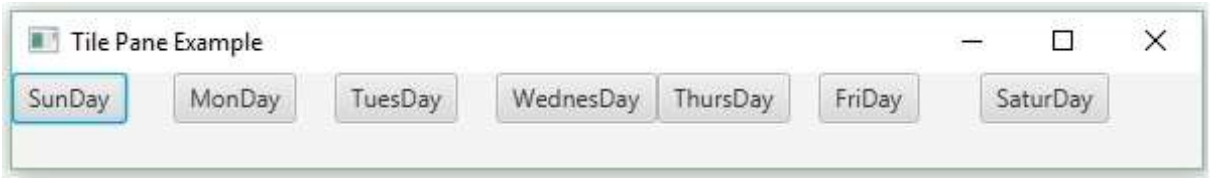
```
            new Button("MonDay"),
            new Button("TuesDay"),
            new Button("WednesDay"),
            new Button("ThursDay"),
            new Button("FriDay"),
            new Button("SaturDay")
        };


        //Creating a Tile Pane
        TilePane tilePane = new TilePane();

        //Setting the orientation for the Tile Pane
        tilePane.setOrientation(Orientation.HORIZONTAL);

        //Setting the allignment for the Tile Pane
        tilePane.setTileAlignment(Pos.CENTER_LEFT);

        //Setting the preferred columns for the Tile Pane
        tilePane.setPrefRows(4);

        //Retrieving the observable list of the Anchor Pane
        ObservableList list = tilePane.getChildren();

        //Adding the array of buttons to the pane
        list.addAll(buttons);

        //Creating a scene object
        Scene scene = new Scene(tilePane);

        //Setting title to the Stage
        stage.setTitle("Tile Pane Example");

        //Adding scene to the satge
        stage.setScene(scene);
```

```
      //Displaying the contents of the stage
      stage.show();
   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac TilePaneExample.java
java TilePaneExample
```

On executing, the above program generates a JavaFX window as shown below.



GridPane

If we use Grid Pane in our application, all the nodes that are added to it are arranged in a way that they form a grid of rows and columns. This layout comes handy while creating forms using JavaFX.

The class named **GridPane** of the package **javafx.scene.layout** represents the GridPane. This class provides eleven properties, which are –

- **alignment:** This property represents the alignment of the pane and you can set value of this property using the **setAlignment()** method.
- **hgap:** This property is of the type double and it represents the horizontal gap between columns.
- **vgap:** This property is of the type double and it represents the vertical gap between rows.
- **gridLinesVisible:** This property is of Boolean type. On true, the lines of the pane are set to be visible.

Following are the cell positions in the grid pane of JavaFX –

(0, 0)	**(1, 0)**	**(2, 0)**

(0, 1)	(1, 1)	(2, 1)
(0, 2)	(1, 2)	(2, 2)

Example

The following program is an example of the grid pane layout. In this, we are creating a form using a Grid Pane.

Save this code in a file with the name **GridPaneExample.java.**

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.GridPane;
import javafx.scene.text.Text;
import javafx.scene.control.TextField;
import javafx.stage.Stage;
public class GridPaneExample extends Application {

   @Override
   public void start(Stage stage) {

      //creating label email
      Text text1 = new Text("Email");
      //creating label password
      Text text2 = new Text("Password");

      //Creating Text Filed for email
      TextField textField1 = new TextField();
      //Creating Text Filed for password
      TextField textField2 = new TextField();

      //Creating Buttons
      Button button1 = new Button("Submit");
      Button button2 = new Button("Clear");
```

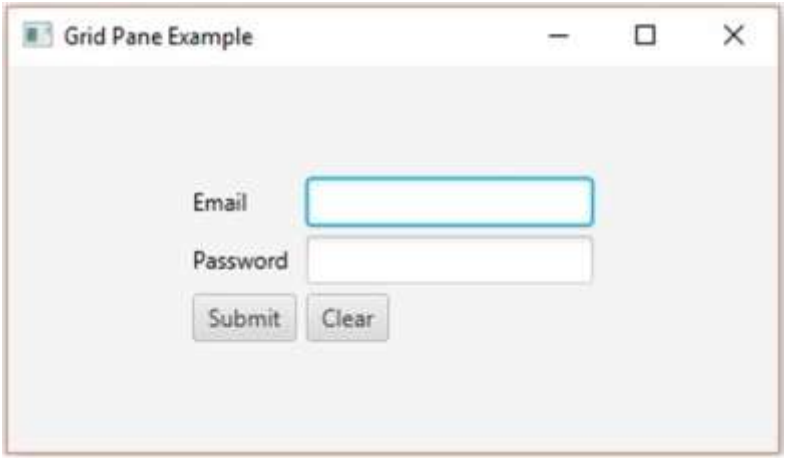
```
      //Creating a Grid Pane
      GridPane gridPane = new GridPane();
      //Setting size for the pane

      gridPane.setMinSize(400, 200);

      //Setting the padding
      gridPane.setPadding(new Insets(10, 10, 10, 10));
      //Setting the vertical and horizontal gaps between the columns
      gridPane.setVgap(5);
      gridPane.setHgap(5);
      //Setting the Grid allignment
      gridPane.setAlignment(Pos.CENTER);

      //Arrianging all the nodes in the grid
      gridPane.add(text1, 0, 0);
      gridPane.add(textField1, 1, 0);
      gridPane.add(text2, 0, 1);
      gridPane.add(textField2, 1, 1);
      gridPane.add(button1, 0, 2);
      gridPane.add(button2, 1, 2);

      //Creating a scene object
      Scene scene = new Scene(gridPane);

      //Setting title to the Stage
      stage.setTitle("Grid Pane Example");

      //Adding scene to the satge
      stage.setScene(scene);

      //Displaying the contents of the stage
      stage.show();
   }
   public static void main(String args[]){
      launch(args);
```

```
    }
}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac GridPaneExample.java
java GridPaneExample
```

On executing, the above program generates a JavaFX window as shown below.



17. JavaFX – CSS

Cascading Style Sheets, also referred to as **CSS**, is a simple design language intended to simplify the process of making web pages presentable.

CSS handles the look and feel part of a web page. Using CSS, you can control the color of the text, style of fonts, spacing between paragraphs, size of columns and layout. Apart from these, you can also control the background images or colors that are used, layout designs, variations in display for different devices and screen sizes as well as a variety of other effects.

CSS in JavaFX

JavaFX provides you the facility of using CSS to enhance the look and feel of the application. The package **javafx.css** contains the classes that are used to apply CSS for JavaFX applications.

A CSS comprises of style rules that are interpreted by the browser and then applied to the corresponding elements in your document.

A style rule is made of three parts, which are –

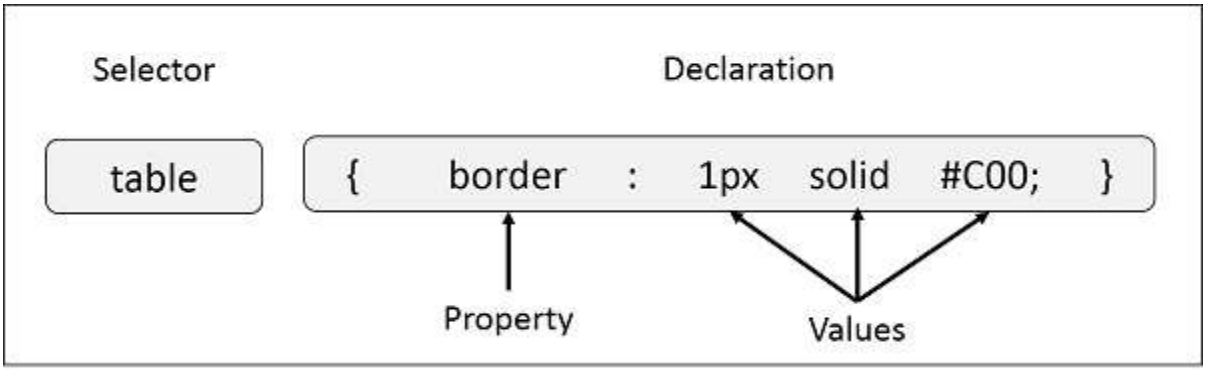
- **Selector** – A selector is an HTML tag at which a style will be applied. This could be any tag like **<h1>** or **<table>**, etc.

- **Property** – A property is a type of attribute of the HTML tag. In simpler terms, all the HTML attributes are converted into CSS properties. They could be **color**, **border**, etc.

- **Value** - Values are assigned to properties. For example, a color property can have value either **red** or **#F1F1F1**, etc.

You can put CSS Style Rule Syntax as follows –

```
selector { property: value }
```



The default style sheet used by JavaFX is **modena.css**. It is found in the JavaFX runtime jar.

Adding Your own Style Sheet

You can add your own style sheet to a scene in JavaFX as follows –

```
Scene scene = new Scene(new Group(), 500, 400);
scene.getStylesheets().add("path/stylesheet.css");
```

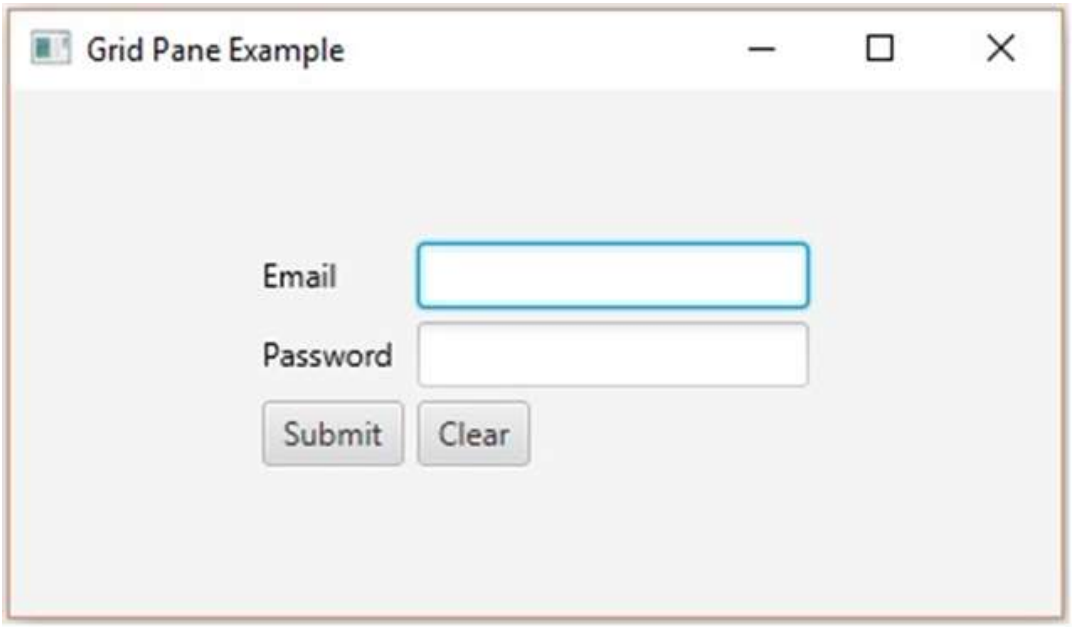
Adding Inline Style Sheets

You can also add in-line styles using the **setStyle()** method. These styles consist of only key-value pairs and they are applicable to the nodes on which they are set. Following is a sample code of setting an inline style sheet to a button.

```
.button {
-fx-background-color: red;
-fx-text-fill: white;
}
```

Example

Assume that we have developed an JavaFX application which displays a form with a Text Field, Password Field, Two Buttons. By default, this form looks as shown in the following screenshot:



The following program is an example which demonstrates how to add styles to the above application in JavaFX.

Save this code in a file with the name **CssExample.java**

```
import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.PasswordField;
import javafx.scene.layout.GridPane;
import javafx.scene.text.Text;
import javafx.scene.control.TextField;
import javafx.stage.Stage;

public class CssExample extends Application {

   @Override
   public void start(Stage stage) {

      //creating label email
      Text text1 = new Text("Email");
      //creating label password
      Text text2 = new Text("Password");

      //Creating Text Filed for email
      TextField textField1 = new TextField();
      //Creating Text Filed for password
      PasswordField textField2 = new PasswordField();

      //Creating Buttons
      Button button1 = new Button("Submit");
      Button button2 = new Button("Clear");

      //Creating a Grid Pane
      GridPane gridPane = new GridPane();
      //Setting size for the pane
      gridPane.setMinSize(400, 200);
```

```
      //Setting the padding
      gridPane.setPadding(new Insets(10, 10, 10, 10));
      //Setting the vertical and horizontal gaps between the columns
      gridPane.setVgap(5);
      gridPane.setHgap(5);
      //Setting the Grid allignment
      gridPane.setAlignment(Pos.CENTER);

      //Arrianging all the nodes in the grid
      gridPane.add(text1, 0, 0);
      gridPane.add(textField1, 1, 0);
      gridPane.add(text2, 0, 1);
      gridPane.add(textField2, 1, 1);
      gridPane.add(button1, 0, 2);
      gridPane.add(button2, 1, 2);

      //Styling nodes
      button1.setStyle("-fx-background-color: darkslateblue; -fx-text-fill:
white;");
      button2.setStyle("-fx-background-color: darkslateblue; -fx-text-fill:
white;");

      text1.setStyle("-fx-font: normal bold 20px 'serif' ");
      text2.setStyle("-fx-font: normal bold 20px 'serif' ");

      gridPane.setStyle("-fx-background-color: BEIGE;");

      // Creating a scene object
      Scene scene = new Scene(gridPane);

      // Setting title to the Stage
     stage.setTitle("CSS Example");

     // Adding scene to the satge
     stage.setScene(scene);
```

```
      //Displaying the contents of the stage
      stage.show();
   }

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

Compile and execute the saved java file from the command prompt using the following commands.

```
javac CssExample.java
java CssExample
```

On executing, the above program generates a JavaFX window as shown below.