Introduction

JavaFX is a Java library that may be used to create both desktop applications and RIA (Rich Internet Applications). JavaFX applications can run on a variety of platforms, including the web, mobile devices, and desktop computers.

History of JavaFX

Chris Oliver is the creator of JavaFX. Form Follows Functions (F3) was the project's original name. Its purpose is to provide more advanced functionality for GUI application development. In June 2005, Sun Microsystems purchased the F3 project and renamed it JavaFX.

Sun Microsystems made an official announcement during the W3 Conference in 2007.JavaFX 1.0 was released in October 2008. ORACLE Corporation purchased Sun Microsystems in 2009 and launched JavaFX 1.2. JavaFX 1.8, which was released on March 18, 2014, is the most recent version.

Features of JavaFX

The JavaFX library has a number of unique features that make it a top choice for developers creating rich client applications. The following are some of these characteristics/features:

- **Java Library:** JavaFX is a Java library that enables users to take advantage of all of Java's features, including multithreading, generics, lambda expressions, and many others. To write, compile, run, debug, and package their JavaFX application, the user can use any Java editor or IDE of their choosing, such as Eclipse or NetBeans.
- **FXML:** FXML is a declarative markup language based on XML. To provide a more advanced GUI to the user, the coding can be done in FXML.
- **Scene Builder:** Scene Builder, a visual editor for FXML, is another tool implemented by JavaFX. Scene Builder generates FXML markups that can be sent to IDEs such as Eclipse and NetBeans, allowing the user to merge business logic with their applications.
- **Platform Independent:** JavaFX-based rich internet apps are platform-independent. The JavaFX library is available for any scripting language that can run on a JVM, including Java, Groovy, Scala, and JRuby.
- **Web view:** JavaFX applications support the embedding of web pages. Web View embeds web pages using WebKitHTML technology.
- **Hardware-accelerated Graphics Pipeline:** The visuals of JavaFX applications are rendered using the Prism hardware-accelerated graphics rendering pipeline. When used with a compatible graphics card or graphics processing unit, the Prism engine provides smooth JavaFX graphics that may be rendered quickly (GPU).
- **Built-in UI controls:** Built-in components in JavaFX are independent of the operating system. The UI component is sufficient for creating a full-featured application.
- **CSS-like styling:** To improve the style of the application, JavaFX code can be incorporated with CSS. With a basic understanding of CSS, we can improve the appearance of our application.
- **Rich set of APIs:** The JavaFX framework also includes a useful set of APIs for creating GUI applications, 2D and 3D graphics, and much more. This collection of APIs also encompasses all of the Java platform's features. As a result of working with this API, a user can have access to Java language capabilities such as Generics, Annotations, Multithreading, and Lambda Expressions, as well as many more features.
- **High-Performance media engine:** The media pipeline allows for low-latency playback of web multimedia. It uses the Gstreamer Multimedia framework as its foundation.
- **Self-contained application deployment model:** All of the application resources, as well as a private copy of Java and JavaFX Runtime, are included in self-contained application packages.

Architecture of JavaFX

There are various built-in features in JavaFX that are interrelated. The JavaFX library contains a useful set of APIs, classes, and interfaces that are more than enough to create sophisticated online applications and graphical user interfaces that work consistently across different platforms.

The JavaFX architecture is made up of a number of different components. The following is a brief description of these components:

- **JavaFX API:** The top layer of the JavaFX architecture contains the JavaFX public API, which implements all of the essential classes for creating a full-featured JavaFX application with rich graphics.
- **javafx.animation:** It offers classes for combining transition-based animations to JavaFX nodes, such as fill, fade, rotate, scale, and translation.
- **javafx.css:** It consists of classes for applying CSS-like styling to JavaFX GUI applications.
- **javafx.geometry:** It offers classes for representing 2D figures and performing methods on them.
- **javafx.scene:** This JavaFX API package contains classes and interfaces for establishing the scene graph. It also renders sub-packages like canvas, control, input, layout, paint, shape, transform, web, and so on.
- **javafx.application:** This package contains a set of classes that are responsible for the JavaFX application's life cycle.
- **javafx.event :** It contains classes and interfaces for performing and managing JavaFX events.
- **javafx.stage:** The top-level container classes for the JavaFX application are contained in this JavaFX API package.
- **Scene Graph:** The construction of any GUI application begins with the creation of a Scene Graph. All GUI applications in JavaFX are built using only a Scene Graph. The Scene Graph is made up of nodes, which are the building blocks of rich internet applications.
- **Quantum Toolkit:** It connects Prism and GWT and makes them accessible to JavaFX.

LifeCycle of a JavaFX Application

A JavaFX Application class has three life cycle methods in total. These techniques are:

start(): This method is the JavaFX application's entry point procedure, where all of JavaFX's graphics code is written.

init(): This method is a static method that can be extended. The user is unable to create a stage or scenario using this method.

stop(): The **stop()** function, like the **init()** method, is an empty method that can be altered. The user can use this method to write the code that will bring the application to a halt.

So when a user runs a JavaFX application, there are a few tasks that must be performed in a specific order. The order in which a JavaFX application is launched is as follows.

- The application class is first generated as an instance.
- Following that, the **init()** method is invoked.
- Then the **start()** method is called after the **init()** method.
- The launcher waits for the JavaFX application to finish before calling the **stop()** method after calling the **start()** method.

Termination of JavaFX Application

The JavaFX application is halted implicitly when the last window of the JavaFX application is closed. The user can disable this function by calling the static method **setImplicitExit()** with the Boolean value "False". This method should only ever be used in a static situation.

A JavaFX application can also be directly stopped by using one of the two methods, **Platform.exit()** or **system.exit(int)**.

JavaFX Event Handling

When the user interacts with the application nodes in JavaFX, an event happens. There are many references that the user can utilize to construct an event. In order to trigger an event, the user can, for example, use a mouse, press any key on the keyboard, or navigate through any page of the program. As a result, we may argue that events are essentially announcements that inform us that something has occurred on the user's end. An ideal application is one that handles events in the shortest amount of time possible.

Processing of Events in JavaFX:

Events are generally utilized in JavaFX to tell the program about the actions taken by the user. The instrument to achieve the events, route the event to its goal and offer the application the ability to handle the events is implemented by JavaFX.

The class **javafx.event.Event** is provided by JavaFX. This class contains all of the subclasses that describe the different types of events that can be created in JavaFX. Any event is a subclass of the Event class or one of its subclasses.

In JavaFX, there are numerous events, such as MouseEvent, KeyEvent, ScrollEvent, DragEvent, and so on. By inheriting the class **javafx.event.Event**, a user can further customize their unique event.

Different Types of Events:

In general, JavaFX events are divided into the following categories:

Foreground Events: Foreground events are mostly caused by the user's direct interaction with the application's graphical user interface. For instance, tapping a key, selecting an item from a list, scrolling the page, and so on.

Background Events: Background events do not require the user to engage with the program. These events typically occur as a result of operating system interruptions, operation completion, and other factors.

Creating Our first JavaFX Application

Here, we are creating a simple JavaFX application which prints **hello world** on the console on clicking the button shown on the stage.

Step 1: Extend javafx.application.Application and override start()

As we have studied earlier that **start()** method is the starting point of constructing a JavaFX application therefore we need to first override start method of **javafx.application.Application** class. Object of the class **javafx.stage.Stage** is passed into the **start()** method therefore import this class and pass its object into start method. **JavaFX.application.Application** needs to be imported in order to override start method.
The code will look like following.

```
1.  package application;
2.  import javafx.application.Application;
3.  import javafx.stage.Stage;
4.  public class Hello_World extends Application{
5.
6.      @Override
7.      public void start(Stage primaryStage) throws Exception {
8.          // TODO Auto-generated method stub
9.
10.     }
11. }
```

Step 2: Create a Button

A button can be created by instantiating the **javafx.scene.control.Button** class. For this, we have to import this class into our code. Pass the button label text in Button class constructor. The code will look like following.

```
1.  package application;
2.  import javafx.application.Application;
3.  importjavafx.scene.control.Button;
4.  import javafx.stage.Stage;
5.  public class Hello_World extends Application{
6.
7.      @Override
8.      public void start(Stage primaryStage) throws Exception {
9.          // TODO Auto-generated method stub
10.         Buttonbtn1=newButton("Say, Hello World");
11.
12.     }
13.
14. }
```

Step 3: Create a layout and add button to it

JavaFX provides the number of layouts. We need to implement one of them in order to visualize the widgets properly. It exists at the top level of the scene graph and can be seen as a root node. All the other nodes (buttons, texts, etc.) need to be added to this layout.

In this application, we have implemented **StackPane** layout. It can be implemented by instantiating **javafx.scene.layout.StackPane** class. The code will now look like following.

```
1. package application;
2. import javafx.application.Application;
3. import javafx.scene.control.Button;
4. import javafx.stage.Stage;
5. import javafx.scene.layout.StackPane;
6. public class Hello_World extends Application{
7.
8.     @Override
9.     public void start(Stage primaryStage) throws Exception {
10.        // TODO Auto-generated method stub
11.        Button btn1=new Button("Say, Hello World");
12.        StackPane root=new StackPane();
13.        root.getChildren().add(btn1);
14.
15.    }
16.
17. }
```

Step 4: Create a Scene

The layout needs to be added to a scene. Scene remains at the higher level in the hierarchy of application structure. It can be created by instantiating **javafx.scene.Scene** class. We need to pass the layout object to the scene class constructor. Our application code will now look like following.

```
1.  package application;
2.  import javafx.application.Application;
3.  import javafx.scene.Scene;
4.  import javafx.scene.control.Button;
5.  import javafx.stage.Stage;
6.  import javafx.scene.layout.StackPane;
7.  public class Hello_World extends Application{
8.
9.      @Override
10.     public void start(Stage primaryStage) throws Exception {
11.         // TODO Auto-generated method stub
12.         Button btn1=new Button("Say, Hello World");
13.         StackPane root=new StackPane();
14.         root.getChildren().add(btn1);
15.         Scene scene=new Scene(root);
16.     }
17.
18. }
```

we can also pass the width and height of the required stage for the scene in the Scene class constructor.

Step 5: Prepare the Stage

javafx.stage.Stage class provides some important methods which are required to be called to set some attributes for the stage. We can set the title of the stage. We also need to call show() method without which, the stage won't be shown. Lets look at the code which describes how can be prepare the stage for the application.

```
1.  package application;
2.  import javafx.application.Application;
3.  import javafx.scene.Scene;
4.  import javafx.scene.control.Button;
5.  import javafx.stage.Stage;
6.  import javafx.scene.layout.StackPane;
7.  public class Hello_World extends Application{
8.
9.      @Override
10.     public void start(Stage primaryStage) throws Exception {
11.         // TODO Auto-generated method stub
12.         Button btn1=new Button("Say, Hello World");
13.         StackPane root=new StackPane();
```

```
        root.getChildren().add(btn1);
        Scene scene=new Scene(root);
        primaryStage.setScene(scene);
        primaryStage.setTitle("First JavaFX Application");
        primaryStage.show();
    }

}
```

Step 6: Create an event for the button

As our application prints hello world for an event on the button. We need to create an event for the button. For this purpose, call **setOnAction()** on the button and define a anonymous class Event Handler as a parameter to the method.

Inside this anonymous class, define a method handle() which contains the code for how the event is handled. In our case, it is printing hello world on the console.

```
package application;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;
import javafx.scene.layout.StackPane;
public class Hello_World extends Application{

    @Override
    publicvoid start(Stage primaryStage) throws Exception {
        // TODO Auto-generated method stub
        Button btn1=new Button("Say, Hello World");
        btn1.setOnAction(new EventHandler<ActionEvent>() {

            @Override
            publicvoid handle(ActionEvent arg0) {
                // TODO Auto-generated method stub
                System.out.println("hello world");
            }
        });
        StackPane root=new StackPane();
        root.getChildren().add(btn1);
        Scene scene=new Scene(root,600,400);
        primaryStage.setScene(scene);
```

```
27.         primaryStage.setTitle("First JavaFX Application");
28.         primaryStage.show();
29.     }
30.
31. }
```

Step 7: Create the main method

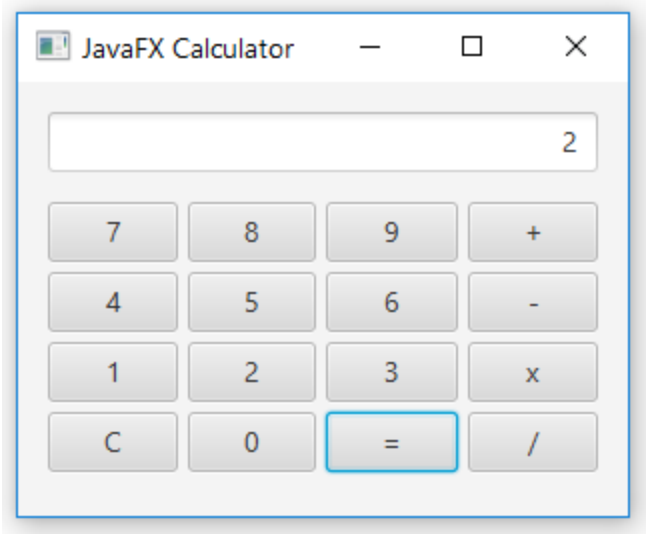
Till now, we have configured all the necessary things which are required to develop a basic JavaFX application but this application is still incomplete. We have not created main method yet. Hence, at the last, we need to create a main method in which we will launch the application i.e. will call launch() method and pass the command line arguments (args) to it. The code will now look like following.

```
1.  package application;
2.  import javafx.application.Application;
3.  import javafx.event.ActionEvent;
4.  import javafx.event.EventHandler;
5.  import javafx.scene.Scene;
6.  import javafx.scene.control.Button;
7.  import javafx.stage.Stage;
8.  import javafx.scene.layout.StackPane;
9.  public class Hello_World extends Application{
10.
11.     @Override
12.     public void start(Stage primaryStage) throws Exception {
13.         // TODO Auto-generated method stub
14.         Button btn1=new Button("Say, Hello World");
15.         btn1.setOnAction(new EventHandler<ActionEvent>() {
16.
17.             @Override
18.             public void handle(ActionEvent arg0) {
19.                 // TODO Auto-generated method stub
20.                 System.out.println("hello world");
21.             }
22.         });
23.         StackPane root=new StackPane();
24.         root.getChildren().add(btn1);
25.         Scene scene=new Scene(root,600,400);
26.         primaryStage.setTitle("First JavaFX Application");
27.         primaryStage.setScene(scene);
28.         primaryStage.show();
29.     }
```

```
30.  publicstaticvoid main (String[] args)
31.  {
32.      launch(args);
33.  }
34.
35. }
```

The Application will produce the following output on the screen.

Example : Calculator




```
 1import javafx.application.Application;
 2import javafx.event.ActionEvent;
 3import javafx.event.EventHandler;
 4import javafx.scene.Scene;
 5import javafx.scene.control.Label;
 6import javafx.scene.control.Button;
 7import javafx.scene.control.TextField;
 8import javafx.scene.layout.BorderPane;
 9import javafx.scene.layout.GridPane;
10import javafx.scene.layout.ColumnConstraints;
11import javafx.scene.layout.Priority;
12import javafx.stage.Stage;
13import javafx.geometry.Insets;
14import javafx.geometry.Pos;
15
16public class JavaFXCalculator extends Application {
17   private TextField tfDisplay;    // display textfield
18   private Button[] btns;          // 16 buttons
19   private String[] btnLabels = {  // Labels of 16 buttons
20      "7", "8", "9", "+",
21      "4", "5", "6", "-",
22      "1", "2", "3", "x",
23      "C", "0", "=", "/"
24   };
25   // For computation
26   private int result = 0;      // Result of computation
27   private String inStr = "0";  // Input number as String
28   // Previous operator: ' '(nothing), '+', '-', '*', '/', '='
29   private char lastOperator = ' ';
30
31   // Event handler for all the 16 Buttons
32   EventHandler handler = evt -> {
33      String currentBtnLabel = ((Button)evt.getSource()).getText();
34      switch (currentBtnLabel) {
35         // Number buttons
```

```
      case "0": case "1": case "2": case "3": case "4":
      case "5": case "6": case "7": case "8": case "9":
        if (inStr.equals("0")) {
          inStr = currentBtnLabel;  // no leading zero
        } else {
          inStr += currentBtnLabel; // append input digit
        }
        tfDisplay.setText(inStr);
        // Clear buffer if last operator is '='
        if (lastOperator == '=') {
          result = 0;
          lastOperator = ' ';
        }
        break;

      // Operator buttons: '+', '-', 'x', '/' and '='
      case "+":
        compute();
        lastOperator = '+';
        break;
      case "-":
        compute();
        lastOperator = '-';
        break;
      case "x":
        compute();
        lastOperator = '*';
        break;
      case "/":
        compute();
        lastOperator = '/';
        break;
      case "=":
        compute();
        lastOperator = '=';
        break;

      // Clear button
      case "C":
        result = 0;
        inStr = "0";
        lastOperator = ' ';
        tfDisplay.setText("0");
        break;
    }
  };

  // User pushes '+', '-', '*', '/' or '=' button.
  // Perform computation on the previous result and the current input number,
  // based on the previous operator.
  private void compute() {
```

```
87      int inNum = Integer.parseInt(inStr);
88      inStr = "0";
89      if (lastOperator == ' ') {
90        result = inNum;
91      } else if (lastOperator == '+') {
92        result += inNum;
93      } else if (lastOperator == '-') {
94        result -= inNum;
95      } else if (lastOperator == '*') {
96        result *= inNum;
97      } else if (lastOperator == '/') {
98        result /= inNum;
99      } else if (lastOperator == '=') {
100       // Keep the result for the next operation
101     }
102     tfDisplay.setText(result + "");
103   }
104
105   // Setup the UI
106   @Override
107   public void start(Stage primaryStage) {
108     // Setup the Display TextField
109     tfDisplay = new TextField("0");
110     tfDisplay.setEditable(false);
111     tfDisplay.setAlignment(Pos.CENTER_RIGHT);
112
113     // Setup a GridPane for 4x4 Buttons
114     int numCols = 4;
115     int numRows = 4;
116     GridPane paneButton = new GridPane();
117     paneButton.setPadding(new Insets(15, 0, 15, 0));  // top, right, bottom, left
118     paneButton.setVgap(5);  // Vertical gap between nodes
119     paneButton.setHgap(5);  // Horizontal gap between nodes
120     // Setup 4 columns of equal width, fill parent
121     ColumnConstraints[] columns = new ColumnConstraints[numCols];
122     for (int i = 0; i < numCols; ++i) {
123       columns[i] = new ColumnConstraints();
124       columns[i].setHgrow(Priority.ALWAYS) ;  // Allow column to grow
125       columns[i].setFillWidth(true);  // Ask nodes to fill space for column
126       paneButton.getColumnConstraints().add(columns[i]);
127     }
128
129     // Setup 16 Buttons and add to GridPane; and event handler
130     btns = new Button[16];
131     for (int i = 0; i < btns.length; ++i) {
132       btns[i] = new Button(btnLabels[i]);
133       btns[i].setOnAction(handler);  // Register event handler
134       btns[i].setMaxSize(Double.MAX_VALUE, Double.MAX_VALUE);  // full-width
135       paneButton.add(btns[i], i % numCols, i / numCols);  // control, col, row
136     }
137
```

```
138      // Setup up the scene graph rooted at a BorderPane (of 5 zones)
139      BorderPane root = new BorderPane();
140      root.setPadding(new Insets(15, 15, 15, 15));  // top, right, bottom, left
141      root.setTop(tfDisplay);     // Top zone contains the TextField
142      root.setCenter(paneButton); // Center zone contains the GridPane of Buttons
143
144      // Set up scene and stage
145      primaryStage.setScene(new Scene(root, 300, 300));
146      primaryStage.setTitle("JavaFX Calculator");
147      primaryStage.show();
148   }
149
150   public static void main(String[] args) {
151      launch(args);
152   }
153}
```

JavaFX 2D Shapes

In some of the applications, we need to show two dimensional shapes to the user. However, JavaFX provides the flexibility to create our own 2D shapes on the screen .

There are various classes which can be used to implement 2D shapes in our application. All these classes resides in **javafx.scene.shape** package.

This package contains the classes which represents different types of 2D shapes. There are several methods in the classes which deals with the coordinates regarding 2D shape creation.

What are 2D shapes?

In general, a two dimensional shape can be defined as the geometrical figure that can be drawn on the coordinate system consist of X and Y planes. However, this is different from 3D shapes in the sense that each point of the 2D shape always consists of two coordinates (X,Y).

Using JavaFX, we can create 2D shapes such as Line, Rectangle, Circle, Ellipse, Polygon, Cubic Curve, quad curve, Arc, etc. The class **javafx.scene.shape.Shape** is the base class for all the shape classes.

How to create 2D shapes?

As we have mentioned earlier that every shape is represented by a specific class of the package **javafx.scene.shape**. For creating a two dimensional shape, the following instructions need to be followed.

1. Instantiate the respective class : for example, **Rectangle rect = new Rectangle()**
2. Set the required properties for the class using instance setter methods: for example,

```
1. rect.setX(10);
2.     rect.setY(20);
3.     rect.setWidth(100);
4.     rect.setHeight(100);
```

3. Add class object to the Group layout: for example,

```
1. Group root = new Group();
2.     root.getChildren().add(rect);
```

The following table consists of the JavaFX shape classes along with their descriptions.

Shape	Description
Line	In general, Line is the geometrical figure which joins two (X,Y) points on 2D coordinate system. In JavaFX, **javafx.scene.shape.Line** class needs to be instantiated in order to create lines.
Rectangle	In general, Rectangle is the geometrical figure with two pairs of two equal sides and four right angles at their joint. In JavaFX, **javafx.scene.shape.Rectangle** class needs to be instantiated in order to create Rectangles.
Ellipse	In general, ellipse can be defined as a curve with two focal points. The sum of the distances to the focal points are constant from each point of the ellipse. In JavaFX. **javafx.scene.shape.Ellipse** class needs to be instantiated in order to create Ellipse.
Arc	Arc can be defined as the part of the circumference of the circle of ellipse. In JavaFX, **javafx.scene.shape.Arc** class needs to be instantiated in order to create Arcs.
Circle	A circle is the special type of Ellipse having both the focal points at the same location. In JavaFX, Circle can be created by instantiating **javafx.scene.shape.Circle** class.

Polygon	Polygon is a geometrical figure that can be created by joining the multiple Co-planner line segments. In JavaFX, **javafx.scene.shape**. Pollygon class needs to be instantiated in order to create polygon.
Cubic Curve	A Cubic curve is a curve of degree 3 in the XY plane. In Javafx, **javafx.scene.shape.CubicCurve** class needs to be instantiated in order to create Cubic Curves.
Quad Curve	A Quad Curve is a curve of degree 2 in the XY plane. In JavaFX, **javafx.scene.shape.QuadCurve** class needs to be instantiated in order to create QuadCurve.

JavaFX Shape Properties

All the JavaFX 2D shape classes acquires the common properties defined by **JavaFX.scene.shape.Shape** class. In the following table, we have described the common shape properties.

Property	Description	Setter Methods
fill	Used to fill the shape with a defined paint. This is a object <paint> type property.	setFill(Paint)
smooth	This is a boolean type property. If true is passes then the edges of the shape will become smooth.	setSmooth(boolean)
strokeDashOffset	It defines the distances in the coordinate system which shows the shapes in the dashing patterns. This is a double type property.	setStrokeDashOffset(Double)
strokeLineCap	It represents the style of the line end cap. It is a strokeLineCap type property.	setStrokeLineCap(StrokeLineCap)
strokeLineJoin	It represents the style of the joint of the two paths.	setStrokeLineJoin(StrokeLineJoin)
strokeMiterLimit	It applies the limitation on the distance between the inside and outside points of a joint. It is a double type property.	setStrokeMiterLimit(Double)
stroke	It is a colour type property which represents the colour of the boundary line of the shape.	setStroke(paint)
strokeType	It represents the type of the stroke (where the boundary line will be imposed to the shape) whether inside, outside or centred.	setStrokeType(StrokeType)
strokeWidth	It represents the width of the stroke.	setStrokeWidth(Double)

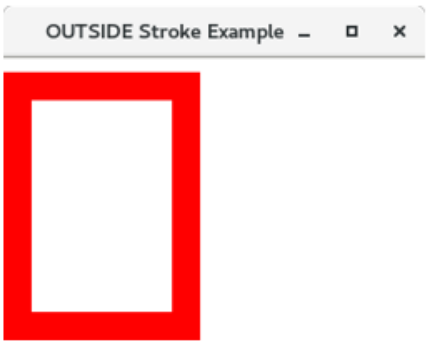
Examples

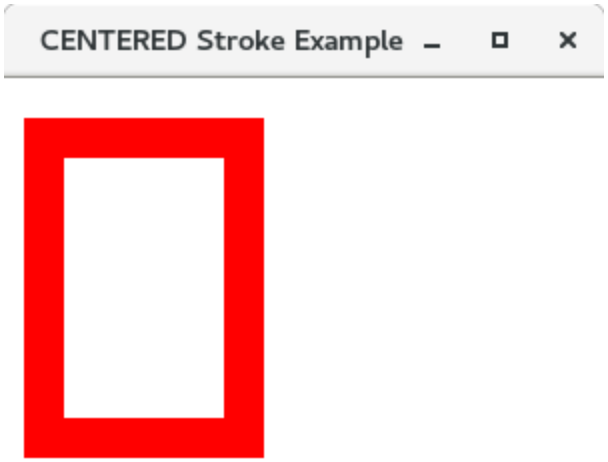

setFill Example


setStroke Example


setStrokeWidth Example


INSIDE Stroke Example

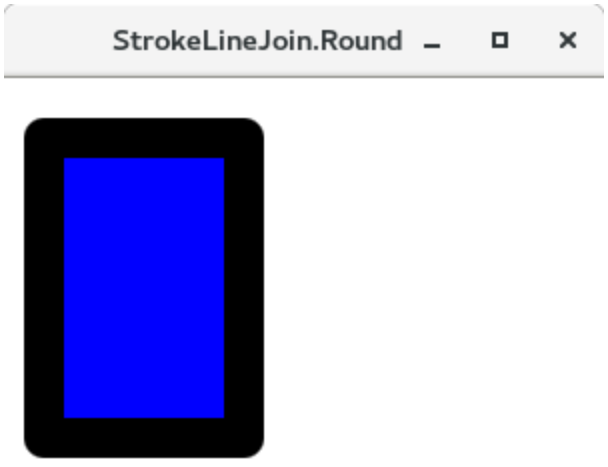

OUTSIDE Stroke Example


CENTERED Stroke Example


setSmooth(true) Example


StrokeLineJoin.BEVEL


StrokeLineJoin.Miter


StrokeLineJoin.Round

JavaFX Line

In general, Line can be defined as the geometrical structure which joins two points (X1,Y1) and (X2,Y2) in a X-Y coordinate plane. JavaFX allows the developers to create the line on the GUI of a JavaFX application. JavaFX library provides the class **Line** which is the part of **javafx.scene.shape** package.
Follow the following instructions to create a Line.

- Instantiate the class **javafx.scene.shape.Line**.
- set the required properties of the class object.
- Add class object to the group

Properties

Line class contains various properties described below.

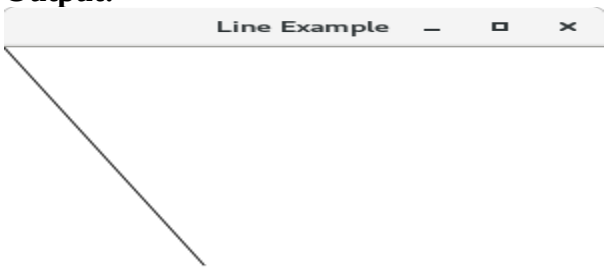
Property	Description	Setter Methods
endX	The X coordinate of the end point of the line	setEndX(Double)
endY	The y coordinate of the end point of the line	setEndY(Double)
startX	The x coordinate of the starting point of the line	setStartX(Double)
startY	The y coordinate of the starting point of the line	setStartY(Double)

Example 1:

```
1. package application;
2. import javafx.application.Application;
3. import javafx.scene.Scene;
4. import javafx.scene.Group;
5. import javafx.scene.shape.Line;
6. import javafx.stage.Stage;
7. public class LineDrawingExamples extends Application{
8.
9.     @Override
10.    public void start(Stage primaryStage) throws Exception {
11.        // TODO Auto-generated method stub
12.        Line line = new Line(); //instantiating Line class
13.        line.setStartX(0); //setting starting X point of Line
14.        line.setStartY(0); //setting starting Y point of Line
15.        line.setEndX(100); //setting ending X point of Line
16.        line.setEndY(200); //setting ending Y point of Line
17.        Group root = new Group(); //Creating a Group
18.        root.getChildren().add(line); //adding the class object //to the group
19.        Scene scene = new Scene(root,300,300);
20.        primaryStage.setScene(scene);
```

```
21.         primaryStage.setTitle("Line Example");
22.         primaryStage.show();
23.
24.     }
25.     public static void main(String[] args) {
26.         launch(args);
27.     }
28.
29. }
```

Output:



JavaFX Ellipse

In general, ellipse can be defined as the geometrical structure with the two focal points. The focal points in the ellipse are chosen so that the sum of the distance to the focal points is constant from every point of the ellipse.
In JavaFX, the class **javafx.scene.shape.Ellipse** represents Ellipse. This class needs to be instantiated in order to create ellipse. This class contains various properties which needs to be set in order to render ellipse on a XY place.

Properties

Property	Description	Setter Methods
CenterX	Horizontal position of the centre of eclipse	setCenterX(Double X-value)
CenterY	Vertical position of the centre of eclipse	setCenterY(Double Y-value)
RadiousX	Width of Eclipse	setRadiusX(Double X-Radious Vaue)
RadiousY	Height of Eclipse	setRadiusY(Double Y-Radious Value)

How to create Ellipse?

There are the three main steps which needs to be followed in order to create ellipse

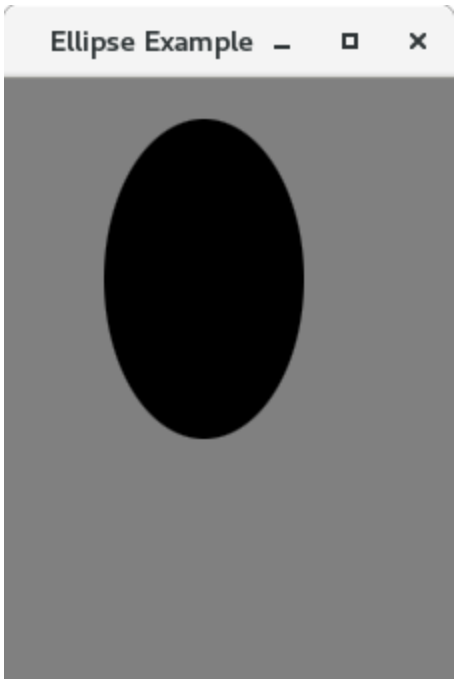
- Instantiate Ellipse class.
- Set the requires properties for the class.
- Add the class object to the group.

Example

```
1. package application;
2. import javafx.application.Application;
```

```
3. import javafx.scene.Group;
4. import javafx.scene.Scene;
5. import javafx.scene.paint.Color;
6. import javafx.scene.shape.Ellipse;
7. import javafx.stage.Stage;
8. public class Shape_Example extends Application{
9.
10.     @Override
11.     public void start(Stage primaryStage) throws Exception {
12.         // TODO Auto-generated method stub
13.     primaryStage.setTitle("Ellipse Example");
14.     Group group = new Group();
15.     Ellipse elipse = new Ellipse();
16.     elipse.setCenterX(100);
17.     elipse.setCenterY(100);
18.     elipse.setRadiusX(50);
19.     elipse.setRadiusY(80);
20.     group.getChildren().addAll(elipse);
21.     Scene scene = new Scene(group,200,300,Color.GRAY);
22.     primaryStage.setScene(scene);
23.     primaryStage.show();
24. }
25. public static void main(String[] args) {
26.     launch(args);
27. }
28.
29. }
```

Output:



JavaFX Arc

In general, Arc is the part of the circumference of a circle or ellipse. It needs to be created in some of the JavaFX applications wherever required. JavaFX allows us to create the Arc on GUI by just instantiating **javafx.scene.shape.Arc** class. Just set the properties of the class to the appropriate values to show arc as required by the Application.

Properties

JavaFX Arc properties and their setter method are given in the table below.

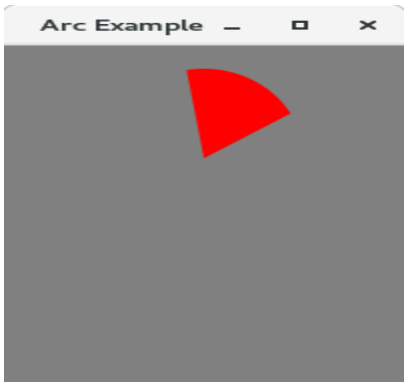
Property	Description	Method
CenterX	X coordinate of the centre point	serCenterX(Double value)
CenterY	Y coordinate of the centre point	setCenterY(Double value)
Length	Angular extent of the arc in degrees	setLength(Double value)
RadiousX	Full width of the ellipse of which, Arc is a part.	setRadiusX(Double value)
RadiousY	Full height of the ellipse of which, Arc is a part	setRadiusY(Double value)
StartAngle	Angle of the arc in degrees	setStartAngle(Double value)
type	Type of Arc : OPEN, CHORD, ROUND	setType(Double value)

Example

```
1. package application;
2. import javafx.application.Application;
3. import javafx.scene.Group;
4. import javafx.scene.Scene;
5. import javafx.scene.paint.Color;
```

```
6.  import javafx.scene.shape.Arc;
7.  import javafx.scene.shape.ArcType;
8.  import javafx.stage.Stage;
9.  public class Shape_Example extends Application{
10.
11.     @Override
12.     public void start(Stage primaryStage) throws Exception {
13.         // TODO Auto-generated method stub
14.     primaryStage.setTitle("Arc Example");
15.     Group group = new Group();
16.     Arc arc = new Arc();
17.     arc.setCenterX(100);
18.     arc.setCenterY(100);
19.     arc.setRadiusX(50);
20.     arc.setRadiusY(80);
21.     arc.setStartAngle(30);
22.     arc.setLength(70);
23.     arc.setType(ArcType.ROUND);
24.     arc.setFill(Color.RED);
25.     group.getChildren().addAll(arc);
26.     Scene scene = new Scene(group,200,300,Color.GRAY);
27.     primaryStage.setScene(scene);
28.     primaryStage.show();
29. }
30. public static void main(String[] args) {
31.     launch(args);
32. }
33.
34. }
```

Output:



JavaFX Cirlce

A circle is a special type of ellipse with both of the focal points at the same position. Its horizontal radius is equal to its vertical radius. JavaFX allows us to create Circle on the GUI of any application by just instantiating **javafx.scene.shape.Circle** class. Just set the class properties by using the instance setter methods and add the class object to the Group.

Properties

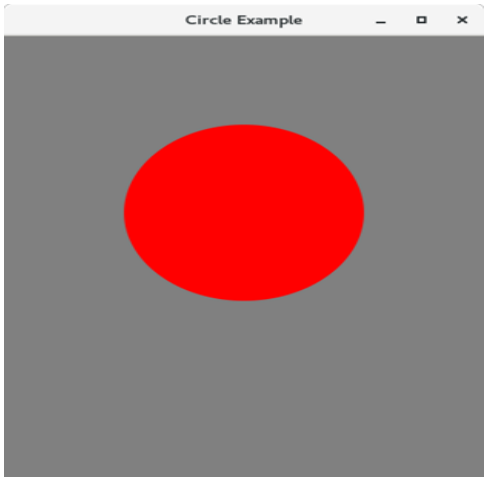
The class properties along with the setter methods and their description are given below in the table.

Property	Description	Setter Methods
centerX	X coordinate of the centre of circle	setCenterX(Double value)
centerY	Y coordinate of the centre of circle	setCenterY(Double value)
radious	Radius of the circle	setRadius(Double value)

Example:

```
1. package application;
2. import javafx.application.Application;
3. import javafx.scene.Group;
4. import javafx.scene.Scene;
5. import javafx.scene.paint.Color;
6. import javafx.scene.shape.Circle;
7. import javafx.stage.Stage;
8. public class Shape_Example extends Application{
9.
10.    @Override
11.    public void start(Stage primaryStage) throws Exception {
12.        // TODO Auto-generated method stub
13.    primaryStage.setTitle("Circle Example");
14.    Group group = new Group();
15.    Circle circle = new Circle();
16.    circle.setCenterX(200);
17.    circle.setCenterY(200);
18.    circle.setRadius(100);
19.    circle.setFill(Color.RED);
20.    group.getChildren().addAll(circle);
21.    Scene scene = new Scene(group,400,500,Color.GRAY);
22.    primaryStage.setScene(scene);
23.    primaryStage.show();
24. }
25. public static void main(String[] args) {
```

```
26.     launch(args);
27. }
28.
29. }
```



JavaFX Polygons

Polygon can be defined as a plain figure with at least three straight sides forming a loop. In the case of polygons, we mainly considers the length of its sides and the interior angles. Triangles, squares, Pentagons, Hexagons,etc are all polygons.

In JavaFX, Polygon can be created by instantiating **javafx.scene.shape.Polygon** class. We need to pass a Double array into the class constructor representing X-Y coordinates of all the points of the polygon. The syntax is given below.

```
1. Polygon poly = new Polygon(DoubleArray);
```

We can also create polygon by anonymously calling **addAll()** method on the reference returned by calling **getPoints()** method which is an instance method of Polygon class. However, we need to pass the double array into this method, which represents X-Y coordinates of the polygon. The syntax is given below.

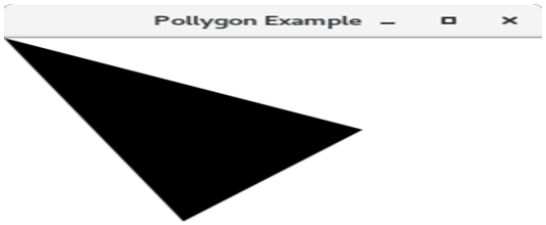
```
1. Pollygon polygon_object = new Pollygon();
2. Pollygon_Object.getPoints().addAll(Double_Array);
```

Example:

The following example creates a polygon with three sides.

```
1. package application;
2. import javafx.application.Application;
3. import javafx.scene.Group;
4. import javafx.scene.Scene;
5. import javafx.scene.shape.Polygon;
6. import javafx.stage.Stage;
7. public class Shape_Example extends Application {
8.
9.     @Override
```

```
10.  public void start(Stage primarystage) {
11.    Group root = new Group();
12.    primarystage.setTitle("Pollygon Example");
13.
14.    Polygon polygon = new Polygon();
15.    polygon.getPoints().addAll(new Double[]{
16.        0.0, 0.0,
17.        100.0, 200.0,
18.        200.0, 100.0 });
19.
20.    root.getChildren().add(polygon);
21.    Scene scene = new Scene(root,300,400);
22.    primarystage.setScene(scene);
23.    primarystage.show();
24.  }
25.
26.  public static void main(String[] args) {
27.    launch(args);
28.  }
29. }
```



JavaFX Cubic Curve

In general, cubic curve is a curve of order 3. In JavaFX, we can create cubic curve by just instantiating **javafx.scene.shape.CubicCurve** class. The class contains various properties defined in the table along with the setter methods. These properties needs to be set in order to create the cubic curve as required.

Properties

The properties of Cubic Curve class have the following properties.

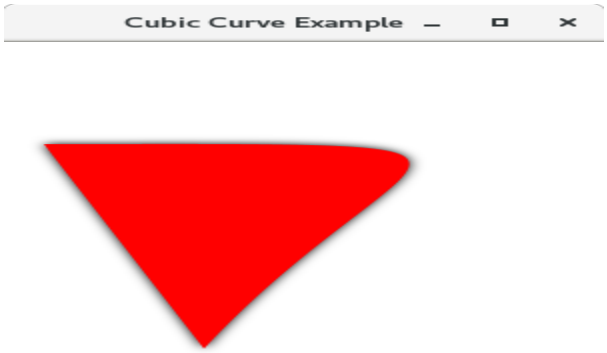
Property	Description	Setter method

controlX1	X coordinate of first control point of cubic curve.	setControlX1(Double)
controlX2	X coordinate of second control point of cubic curve	setControlX2(Double)
controlY1	Y coordinate of first control point of cubic curve	setControlY1(Double)
controlY2	Y coordinate of second control point of cubic curve	setControlX1(Double)
endX	X coordinate of end point of cubic curve	setEndX(Double)
endY	Y coordinate of end point of cubic curve.	setEndY(Double)
startX	X coordinate of starting point of cubic curve	setStartX(Double)
startY	Y coordinate of starting point of cubic curve	setStartY(Double)

Example:

```
1.  package application;
2.  import javafx.application.Application;
3.  import javafx.scene.Group;
4.  import javafx.scene.Scene;
5.  import javafx.scene.effect.DropShadow;
6.  import javafx.scene.paint.Color;
7.  import javafx.scene.shape.CubicCurve;
8.  import javafx.stage.Stage;
9.  public class Shape_Example extends Application {
10.
11.  @Override
12.  public void start(Stage primarystage) {
13.    Group root = new Group();
14.    primarystage.setTitle("Cubic Curve Example");
15.    CubicCurve c = new CubicCurve();
16.    c.setStartX(20);
17.    c.setStartY(100);
18.    c.setControlX1(300);
19.    c.setControlX2(200);
20.    c.setControlY1(100);
21.    c.setControlY2(90);
22.    c.setFill(Color.RED);
23.    c.setEffect(new DropShadow());
24.    c.setEndX(100);
25.    c.setEndY(300);
26.    root.getChildren().add(c);
```

```
27.     Scene scene = new Scene(root,300,400);
28.     primarystage.setScene(scene);
29.     primarystage.show();
30.   }
31.
32.   public static void main(String[] args) {
33.     launch(args);
34.   }
35. }
```



JavaFX Quad Curve

Quad curve is a plain curve of order two. This is different from Cubic Curve in the sense that it doesn't have two control points like cubic curve. It rather has a single control point (X,Y).

In JavaFX, we can instantiate **javafx.scene.shape.QuadCurve** class to create a Quad curve. The class contains various properties which are defined in the table below. The class also contains the setter methods which can be used to set the properties to get the quad curve according to our requirements.

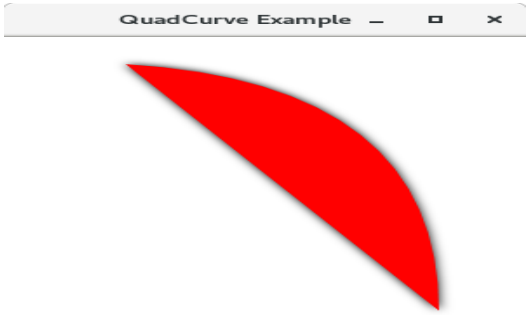
Properties

The properties of the class along with their setter methods are given in the table below.

Property	Description	Setter Method
controlX	X coordinate of the control point of quad curve	setControlX(Double)
controlY	Y coordinate of the control point of quad curve	setControlY(Double)
endX	X coordinate of the end point of quad curve	setEndX(Double)
endY	Y coordinate of ending point of quad curve	setEndY(Double)
startX	X coordinate of starting point of quad curve	setStartX(Double)
startY	Y coordinate of starting point of quad curve	setStartY(Double)

Example

```
1.  package application;
2.  import javafx.application.Application;
3.  import javafx.scene.Group;
4.  import javafx.scene.Scene;
5.  import javafx.scene.effect.DropShadow;
6.  import javafx.scene.paint.Color;
7.  import javafx.scene.shape.QuadCurve;
8.  import javafx.stage.Stage;
9.  public class Shape_Example extends Application {
10.
11.  @Override
12.  public void start(Stage primarystage) {
13.    Group root = new Group();
14.    primarystage.setTitle("QuadCurve Example");
15.    QuadCurve c = new QuadCurve();
16.    c.setStartX(70);
17.    c.setStartY(30);
18.    c.setControlX(250);
19.    c.setControlY(50);
20.    c.setFill(Color.RED);
21.    c.setEffect(new DropShadow());
22.    c.setEndX(250);
23.    c.setEndY(300);
24.    root.getChildren().add(c);
25.    Scene scene = new Scene(root,300,400);
26.    primarystage.setScene(scene);
27.    primarystage.show();
28.  }
29.
30.  public static void main(String[] args) {
31.    launch(args);
32.  }
33. }
```



JavaFX Gradient Color

In Computer Graphics, Gradient Colors (sometimes called **Color Progression)** are used to specify the position dependent colors to fill a particular region. The value of the gradient color varies with the position. Gradient colors produces the smooth color transitions on the region by varying the color value continuously with the position. JavaFX enables us to implement two types of Gradient color transitions:

1. Linear Gradient
2. Radial Gradient

Linear Gradient

To apply linear gradient patterns to the shapes, we need to instantiate the LinearGradient class. This class contains several instance methods described below in the table.

Instance Methods

Type	Method	Description
Boolean	equals(Object o)	Compares two objects
CycleMethod	getCycleMethod()	Defines which cycle method has been applied to LinearGradient.
Double	getEndX()	X coordinate of gradient axis end point
Double	getEndY()	Y coordinate of gradient axis end point
Double	getStartX()	X coordinate of gradient axis start point
Double	getStartY()	Y coordinate of gradient axis start point
List<Stop>	getStops()	Defines the way of distributions of colors along the gradient
Int	hashCode()	Returns hash code for the linear gradient object
Boolean	isOpaque()	Check whether the paint is completely opaque or not.
Boolean	isProprtional()	Checks whether the start and end locations are proportional or not.
String	toString()	Convert Gradient object to string.

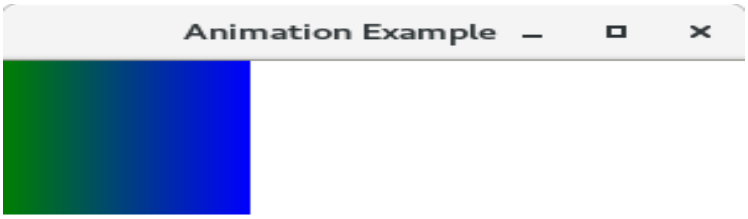
Constructors

The Constructor of this class accepts five parameters:
new LinearGradient(startX, startY, endX, endY, Proportional, CycleMethod, stops)
(startX,startY): represents x and y coordinates of the starting point of the gradient color.
(endX,endY): represents x and y coordinates of the ending point of the gradient color.
Proportional: This is a boolean type property. If this is true then the starting and ending point of the gradient color will become proportional.
CycleMethod: This defines the cycle method applied to the gradient.
Stops: this defines the color distribution along the gradient.

Example:

```
1.  package application;
2.  import javafx.application.Application;
3.  import javafx.scene.Scene;
4.  import javafx.scene.layout.VBox;
5.  import javafx.scene.paint.Color;
6.  import javafx.scene.paint.CycleMethod;
7.  import javafx.scene.paint.LinearGradient;
8.  import javafx.scene.paint.Stop;
9.  import javafx.scene.shape.Rectangle;
10. import javafx.stage.Stage;
11. public class Shape_Example extends Application {
12.
13.    @Override
14.    public void start(Stage primaryStage) {
15.        VBox root = new VBox();
16.        final Scene scene = new Scene(root,300, 250);
17.        Stop[] stops = new Stop[] { new Stop(0, Color.GREEN), new Stop(1, Color.BLUE)};
18.        LinearGradient linear = new LinearGradient(0, 0, 1, 0, true, CycleMethod.NO_CYCLE, stops);
19.
20.        Rectangle rect = new Rectangle(0, 0, 100, 100);
21.        rect.setFill(linear);
22.
23.        root.getChildren().add(rect);
24.
25.        primaryStage.setScene(scene);
26.        primaryStage.setTitle("Animation Example");
27.        primaryStage.show();
28.    }
29.
30.    public static void main(String[] args) {
```

```
31.         launch(args);
32.     }
33.     }
```



JavaFX Effects

Effects are basically actions that can improve the appearance of the graphics. JavaFX provides the package named as **javafx.scene.effect** which contains various classes that can be used to apply effects on the UI graphic components like images and shapes. The effects along with their description is given in the table below.

SN	Effect	Description
1	ColorAdjust	This Effect adjusts the color of the node by varying the properties like Hue, Saturation, Brightness, contrast etc. **javafx.scene.effect.ColorAdjust** class deals with all the stuff regarding adjustments of colors of the node.
2	ColorInput	**javafx.scene.ColorInput** class represents ColorInput effect. It makes a coloured rectangle. This displays a rectangular box if applied to a node.
3	ImageInput	**ImageInput** effect is used to bind the image to the scene. It basically passes the specified image to some effect.
4	Blend	**javafx.scene.effect.Blend** class represents the blend effect. This effect joins the pixels of two inputs and produces the combined output at the same location. There are various blend modes defined in the class which can alter the output appearance.
5	Bloom	**javafx.scene.effect.Bloom** class represents bloom effect. This effect makes the pixels of a few portions of the component glow.
6	Glow	This effect is very much similar to Bloom. This can make the input image glow by enhancing the brightness of the bright pixels.

7	BoxBlur	The blur makes the image unclear. JavaFX provides the class **javafx.scene.effect.BoxBlur** which needs to be instantiated in order to apply the blur effect to the nodes. The Box filter is used in the case of BoxBlur effect in JavaFX.
8	GaussionBlur	In JavaFX, GaussionBlur is used to blur the nodes. This class uses Gaussion Convolution Kernel for this purpose.
9	MotionBlur	**MotionBlur** effect is used to make the nodes blur. By applying this effect, the nodes are seemed to be blurred as they are in motion. **javafx.scene.effect.MotionBlur** class represents this effect.
10	Reflection	It adds the reflection of the node on the bottom of the node. The class named as **javafx.scene.effect.Reflection** represents Reflection effect.
11	SepiaTone	SapiaTone effect makes the node toned with radish brown color. The class named as **javafx.scene.effect.SepiaTone** class represents SapiaTone effect. The resulting nodes are similar to antique photographs.
12	Shadow	This duplicates the nodes with the blurry edges. The class named as **javafx.scene.effect.Shadow** represents Shadow effect.
13	DropShadow	This is a high level effect that is used to display the duplicate content behind the original content with the specified color and size.
14	InnerShadow	This effect displays the shadow inside the edges of the nodes to which it is applied.
15	Lighting	This effect is used to lighten the node from a light source. This effect is represented by **javafx.scene.effect.Lighting** class.
16	Light.Distant	It implements lighting on the node from a distant light source. It is represented by **Light.Distant** class.
17	Light.Spot	It implements lighting on a node from a spot light source. It is represented by **Light.Spot** class.
18	Light.Point	It implements lighting on a node from a point light source. It is represented by **Light.Point** class.

JDBC: The Connectivity Model

JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as −

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs).

All of these different executables are able to use a JDBC driver to access a database, and take advantage of the stored data.

JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code.

Pre-Requisite

Before moving further, you need to have a good understanding of the following two subjects −

- Core JAVA Programming
- SQL or MySQL Database

JDBC Driver

JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

1) JDBC-ODBC bridge driver

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.

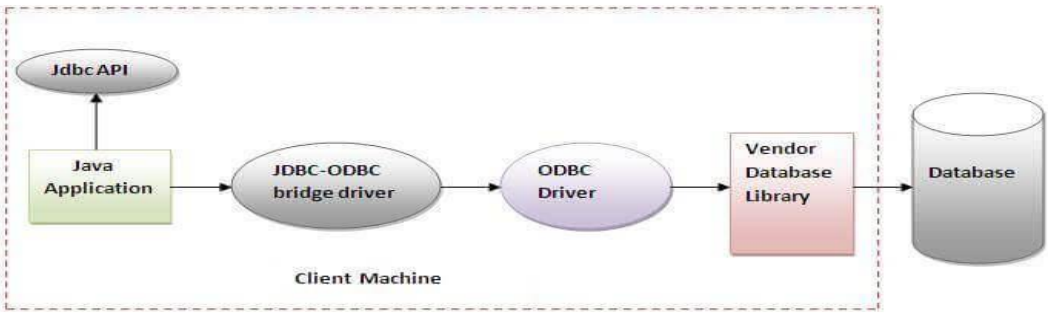


Figure- JDBC-ODBC Bridge Driver

In Java 8, the JDBC-ODBC Bridge has been removed.

Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

Advantages:

- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.
-

2) Native-API driver

The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.

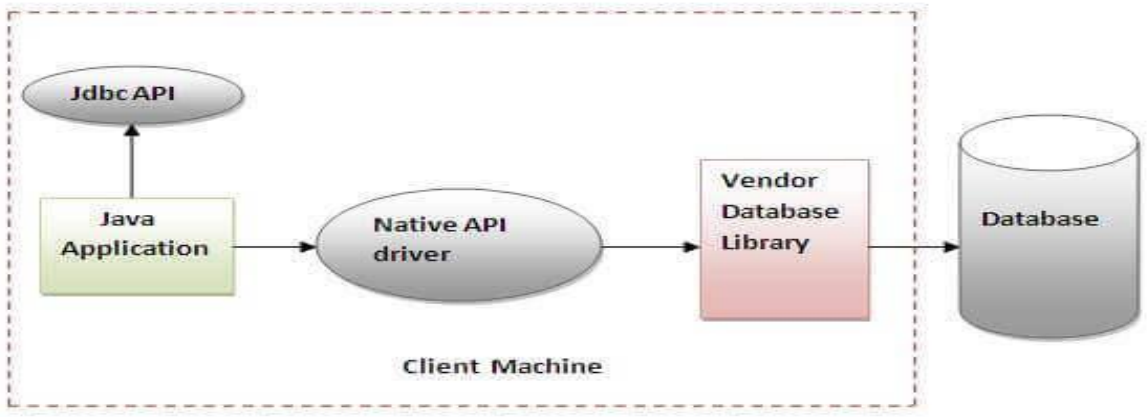


Figure- Native API Driver

Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

3) Network Protocol driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

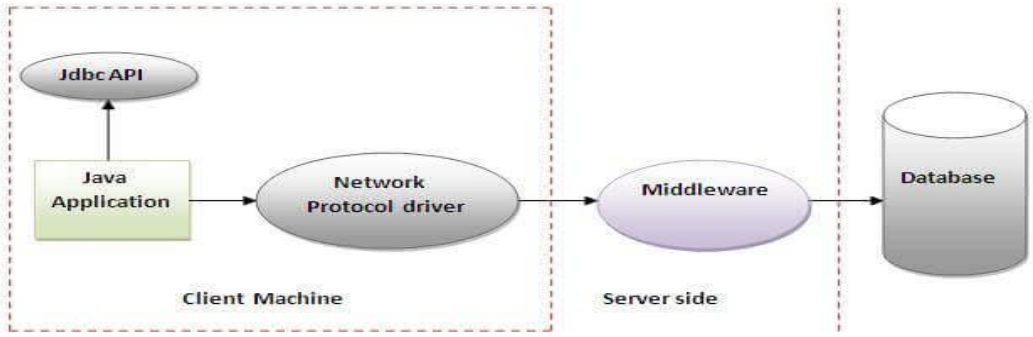


Figure- Network Protocol Driver

Advantage:

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

4) Thin driver

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.

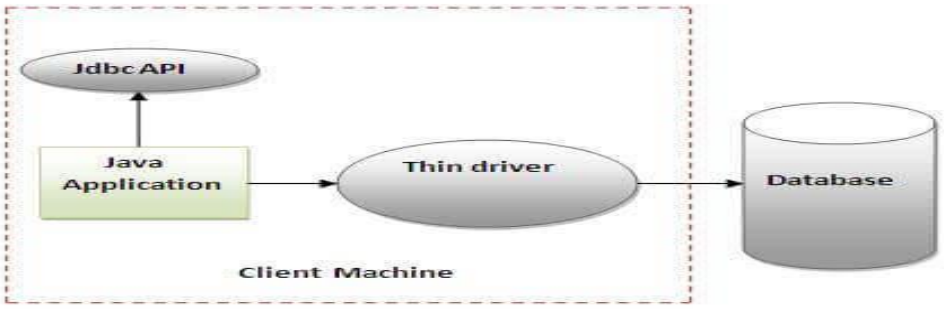


Figure- Thin Driver

Advantage:

- Better performance than all other drivers.
- No software is required at client side or server side.

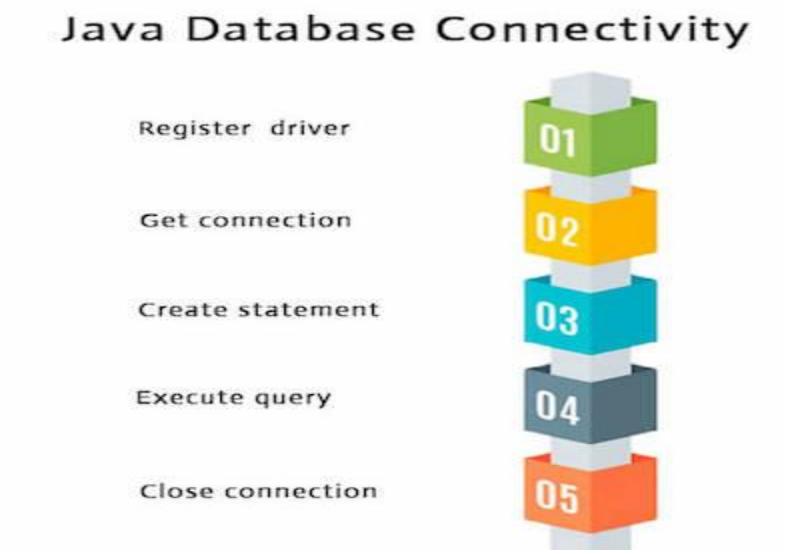
Disadvantage:

- Drivers depend on the Database.

Java Database Connectivity with 5 Steps

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

- Register the Driver class
- Create connection
- Create statement
- Execute queries
- Close connection



1) Register the driver class

The **forName()** method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

Syntax of forName() method

1. **public static void** forName(String className)**throws** ClassNotFoundException

Note: Since JDBC 4.0, explicitly registering the driver is optional. We just need to put vender's Jar in the classpath, and then JDBC driver manager can detect and load the driver automatically.

Example to register the OracleDriver class

Here, Java program is loading oracle driver to esteblish database connection.

1. Class.forName("oracle.jdbc.driver.OracleDriver");

2) Create the connection object

The **getConnection()** method of DriverManager class is used to establish connection with the database.

Syntax of getConnection() method

1. **public static** Connection getConnection(String url)**throws** SQLException
2. **public static** Connection getConnection(String url,String name,String password) **throws** SQLException

Example to establish connection with the Mysql database

```
1. Class.forName("com.mysql.jdbc.Driver");
2. Connection con=DriverManager.getConnection(“jdbc: mysql://localhost:3306/sonoo”,” root”,”root");
```

3) Create the Statement object

The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

Syntax of createStatement() method

1. **public** Statement createStatement()**throws** SQLException

Example to create the statement object

```
1. Statement stmt=con.createStatement();
```

4) Execute the query

The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

Syntax of executeQuery() method

1. **public** ResultSet executeQuery(String sql)**throws** SQLException

Example to execute query

```
1. ResultSet rs=stmt.executeQuery("select * from emp");
2.
3. while(rs.next()){
4. System.out.println(rs.getInt(1)+" "+rs.getString(2));
5. }
```

---

5) Close the connection object

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

Syntax of close() method

1. **public void** close()**throws** SQLException

Example to close connection

```
1. con.close();
```

ODBC vs JDBC

The application programming interfaces (APIs) ODBC (Open Database Connectivity) and JDBC (Java Database Connectivity) are both used to connect to databases. There are, however, some significant distinctions between the two:

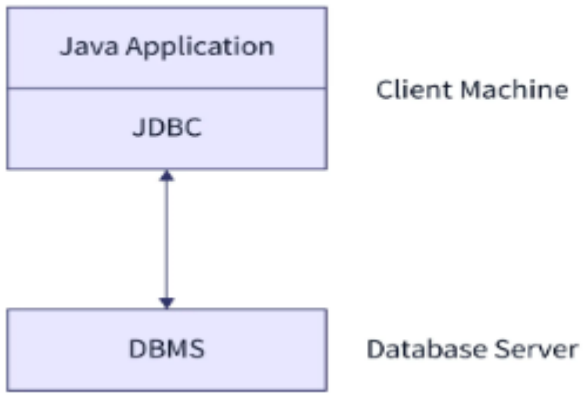
Factors	**ODBC**	**JDBC**
Full-Form	Open Database Connectivity.	Java Database Connectivity.
Language support	ODBC is primarily used for C/C++ applications.	JDBC is specifically designed for Java applications.
API design	ODBC uses a low-level API with a lot of options and functions.	JDBC provides a higher-level API that is easier to use.
Performance	ODBC can be faster than JDBC for some applications.	JDBC is typically faster for Java applications due to its direct integration with the Java language.
Portability	Less Portable.	More Portable.

Types of JDBC Architecture

The JDBC Architecture can be of two types based on the processing models it uses. These models are

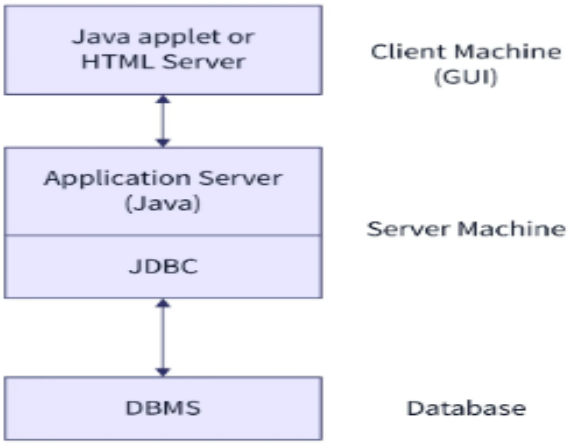
1. 2-tier model
2. 3-tier model

2 Tier Model



- 2-tier JDBC architecture model is a **basic model**.
- In this model, a java application communicates directly to the data sources. JDBC driver is used to establish a connection between the application and the data source.
- When an application needs to interact with a database, a query is directly executed on the data source and the output of the queries is sent back to the user in form of results.
- In this model, the data source can be located on a different machine connected to the same network the user is connected to.
- This model is also known as a **client/server** configuration. Here user's machine acts as a client and the machine on which the database is located acts as a server.

3 Tier Model



- 3-tier model is a **complex and more secure model** of JDBC architecture in java.
- In this model the user queries are sent to the middle tier and then they are executed on the data source.
- Here, the java application is considered as one tier connected to the data source(3rd tier) using middle-tier services.
- In this model user queries are sent to the data source using middle-tier services, from where the commands are again sent to databases for execution.
- The results obtained on the database are again sent to the middle-tier and then to the user/application.

Interfaces of JDBC API

JDBC API uses various interfaces to establish connections between applications and data sources. Some of the popular interfaces in JDBC API are as follows:

- **Driver interface** - The JDBC Driver interface provided implementations of the abstract classes such as ***java.sql.Connection, Statement, PreparedStatement, Driver, etc.*** provided by the JDBC API.
- **Connection interface** - The connection interface is used to create a connection with the database. ***getConnection()*** method of DriverManager class of the Connection interface is used to get a Connection object.
- **Statement interface** - The Statement interface provides methods to execute SQL queries on the database. ***executeQuery(), executeUpdate()*** methods of the Statement interface are used to run SQL queries on the database.
- **ResultSet interface** - ResultSet interface is used to store and display the result obtained by executing a SQL query on the database. ***executeQuery()*** method of statement interface returns a resultset object.
- **RowSet interface** - RowSet interface is a component of Java Bean. It is a wrapper of ResultSet and is used to keep data in tabular form.
- **ResultSetMetaData interface** - Metadata means data about data. ResultSetMetaData interface is used to get information about the resultset interface. The object of the ResultSetMetaData interface provides metadata of resultset like number of columns, column name, total records, etc.
- **DatabaseMetaData interface** - DatabaseMetaData interface is used to get information about the database vendor being used. It provides metadata like database product name, database product version, total tables/views in the database, the driver used to connect to the database, etc.

Classes of JDBC API

Along with interfaces, JDBC API uses various classes that implement the above interfaces. Methods of these classes in JDBC API are used to create connections and execute queries on databases. A list of most commonly used class in JDBC API are as follows:

- **DriverManager class** - DriverManager class is a member of the ***java.sql*** package. It is used to establish a connection between the database and its driver.
- **Blob class** - A ***java.sql.Blob*** is a binary large object that can store large amounts of binary data, such as images or other types of files. Fields defined as TEXT also hold large amounts of data.
- **Clob class** - The ***java.sql.Clob*** interface of the JDBC API represents the CLOB datatype. Since the Clob object in JDBC is implemented using an SQL locator, it holds a logical pointer to the SQL CLOB (not the data).
- **Types class** - Type class defined and store constants that are used to identify generic SQL types also known as JDBC types.

Java Database Connectivity with MySQL

To connect Java application with the MySQL database, we need to follow 5 following steps.
In this example we are using MySql as the database. So we need to know following informations for the mysql database:

1. **Driver class:** The driver class for the mysql database is **com.mysql.jdbc.Driver**.
2. **Connection URL:** The connection URL for the mysql database is **jdbc:mysql://localhost:3306/sonoo** where jdbc is the API, mysql is the database, localhost is the server name on which mysql is running, we may also use IP address, 3306 is the port number and sonoo is the database name. We may use any database, in such case, we need to replace the sonoo with our database name.
3. **Username:** The default username for the mysql database is **root**.
4. **Password:** It is the password given by the user at the time of installing the mysql database. In this example, we are going to use root as the password.

Let's first create a table in the mysql database, but before creating table, we need to create database first.

```
1. create database sonoo;
2. use sonoo;
3. create table emp(id int(10),name varchar(40),age int(3));
```

Example to Connect Java Application with mysql database

In this example, sonoo is the database name, root is the username and password both.

```
1. import java.sql.*;
2. class MysqlCon{
3. public static void main(String args[]){
4. try{
5. Class.forName("com.mysql.jdbc.Driver");
6. Connection con=DriverManager.getConnection( "jdbc:mysql://localhost:3306/sonoo","root","root");
7. //here sonoo is database name, root is username and password
8. Statement stmt=con.createStatement();
9. ResultSet rs=stmt.executeQuery("select * from emp");
10. while(rs.next())
11. System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
12. con.close();
13. }catch(Exception e){ System.out.println(e);}
14. }
15. }
```

Two ways to load the jar file:

1. Paste the mysqlconnector.jar file in jre/lib/ext folder
2. Set classpath

1) Paste the mysqlconnector.jar file in JRE/lib/ext folder:

Download the mysqlconnector.jar file. Go to jre/lib/ext folder and paste the jar file here.

2) Set classpath:

There are two ways to set the classpath:

- temporary
- permanent

How to set the temporary classpath

open command prompt and write:

1. C:>set classpath=c:\folder\mysql-connector-java-5.0.8-bin.jar;.;

How to set the permanent classpath

Go to environment variable then click on new tab. In variable name write **classpath** and in variable value paste the path to the mysqlconnector.jar file by appending mysqlconnector.jar;.; as C:\folder\mysql-connector-java-5.0.8-bin.jar;

References:

1. https://www3.ntu.edu.sg/home/ehchua/programming/java/Javafx1_intro.html
2. https://www.codingninjas.com/studio/library/introduction-to-javafx
3. https://www.javatpoint.com/javafx-gradient-color
4. https://www.tutorialspoint.com/jdbc/jdbc-introduction.htm
5. https://www.javatpoint.com/jdbc-driver
6. https://www.javatpoint.com/example-to-connect-to-the-mysql-database
7. https://intellipaat.com/blog/java-jdbc/#no1