BACHOLER OF COMPUTER APPLICATIONS

SEMESTER 4

DCA2202

JAVA PROGRAMMING

Unit 12

Java Swing and JavaFX

Table of Contents

1. INTRODUCTION

Java Swing is a set of program components for Java programmers to create graphical user interface, like buttons and scroll bars, which are platform independent. These program components are used with Java Foundation Class (JFC). On other side, 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. In this unit Java Swing and JavaFX Architecture have been discussed with example.

1.1 Objectives

After studying this unit, you should be able to:

- *Describe Java Foundation Classes*
- *Describe Java Swing Packages*
- *Describe Swing Component Classes*
- *Describe Swing Components*
- *Describe the Swing Event Handling*
- *Describe JavaFX – Architecture*
- *Describe JavaFX FXML Application*
- *Describe Layout Pane*

2. JAVA FOUNDATION CLASSES

Java Foundation Classes are a set of Graphical User Interface. JFC adds rich graphics functionality and interactivity to Java applications. Java Foundation Classes are superset that contains AWT and completely written in java. Unlike AWT, Java Swing provides platform-independent and lightweight components.

The classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc. are provided by javax.swing package.

AWT

Although the most powerful and exciting features of the JFC were introduced in the later versions of the Java, the JFC also includes the graphics and GUI features of Java 1.0 and Java 1.1. These features are provided by the Abstract Windowing Toolkit (AWT). The graphics and GUI capabilities of the AWT are rudimentary, and many of them have been superseded by more advanced features in Java. Nevertheless, the AWT is the bedrock upon which more advanced JFC functionality is built.

Swing

Swing is an advanced GUI toolkit written in pure Java. It is built upon the AWT but provides many new GUI components and useful GUI-related application services. Swing offers a pluggable look-and-feel architecture that allows an application to be trivially configured either to display a platform- independent Java look-and-feel or to mimic the look-and-feel of the native operating system. Swing also includes an accessibility API that enables the use of assistive technologies, such as screen readers or screen magnifiers for the vision impaired. Many features of Swing are based on the pioneering design of the Netscape Internet Foundation Classes.

3. JAVA SWING

3.1 Java Swing Packages

Swing, like any well-behaved collection of Java software, groups its classes and interfaces into packages. Swing's packages, and their contents, are:

i. **The Accessibility package:** This package defines a contract between Java UI objects (such as screen readers, Braille terminals, and so on) and screen access products used by people with disabilities. Swing components fully support the accessibility interfaces defined in the accessibility package, making it easy to write programs with Swing that people with disabilities can use. The design of the accessibility package also makes it easy to add accessibility support to custom components. These custom components can extend existing Swing components, can extend existing AWT components, or can be lightweight components developed from scratch.

ii. **The Swing component package (javax.swing):** The Swing component package is the largest of Swing's five packages. As Swing's initial beta release drew near, the Swing component package contained 99 classes and 23 interfaces. With a couple of exceptions, the Swing component package is also the package that implements all the component classes used in Swing. (The exceptions are JTableHeaderapi, implemented in the javax.swing.tableapi package, and JTextComponentapi, implemented in javax.swing.textapi.) Swing's UI classes (those that have names beginning with "J") are classes that are actually used to implement components; all other Swing classes are non-UI classes that provide services and functionalities to applications and components. For more details, see the section headed "Varieties of Swing classes."

iii. **The basic package (javax.swing.plaf.basic):** The basic package is the second largest package in the Swing set. It contains around 57 classes (but no interfaces). This package defines the default look-and- feel characteristics of Swing components. By subclassing the classes in the basic package, you can create components with your own customized pluggable L&F designs.)

iv. **The border package (javax.swing.borderapi):** This package contains one interface and nine classes that you can subclass when you want to draw a specialized border around a component. (You don't have to touch this package when you create components with the default borders prescribed by whatever look and feel you are using. Swing draws default borders automatically around standard components.)

v. **The event package (javax.swing.event):** The event package defines Swing-specific event classes. Its role is similar to that of the java.awt.event package. Under most ordinary conditions, you'll probably never have to subclass the components provided in the event package. But if you ever need to create a component that needs to monitor some other particular component -- perhaps in a special way -- you can tap one of the classes provided in this package. Swing's JListapi class, for example, uses this package to determine when the user selects an item in a list.

vi. **The multi package (javax.swing.plaf.multi):** The multi package contains Swing's Multiplexing UI classes (delegates), which permit components to have UIs provided by multiple user-interface (UI) factories.

vii. **The pluggable L&F package (javax.swing.plaf):** The pluggable L&F package contains the classes that Swing uses to provide its components with the pluggable look-and-feel capabilities. Most developers will never need this package. However, if you're building a new look and feel and you can't take the conventional approach of just subclassing the classes in the swing.plaf.basic package, you can start with the abstract classes in swing.plaf. L&F-specific packages that became available with the release of JFC 1.1 (Swing 1.0) include:

1. javax.swing.plaf.metal.
2. javax.swing.plaf.motif.
3. javax.swing.plaf.windows.

viii. **The table package (javax.swing.tableapi):** The table package contains several low-level classes that Swing uses to help you build, view, and manipulate tables.

ix. **The text package (javax.swing.textapi):** This package contains classes and interfaces that deal with components which contain text.

x. **The HTML package (javax.swing.text.htmlapi):** This package contains an HTMLEditorKitapi class that implements a simple text editor for HTML text files.

xi. **The RTF package (javax.swing.text.rtfapi):** This package implements a simple text editor for RTF (rich text format) files.

xii. **The tree package (javax.swing.treeapi):** The classes in this package are used to create, view, and manage tree components.

xiii. **The undo package (javax.swing.undoapi)**: Developers of packages with undo capabilitie for example, text editors -- may need to know about classes and methods defined in this package.

3.2 Java Swing Example

Let's see a simple swing example (FirstSwingExample.java) where we are creating one button and adding it on the JFrame object inside the main() method. The example is given in the figure 10.1 and figure 10.2 shows the output.

```
import javax.swing.*;

public class FirstSwingExample {
      public static void main (String[] args) {
            JFrame f = new JFrame();
            JButton b = new JButton("Click");
            b.setBounds(130,100,100,40);
            f.add(b);
            f.setSize(400,500);
            f.setLayout(null);
            f.setVisible(true);
      }
}
```

Fig 10.1: FirstSwingExample.java

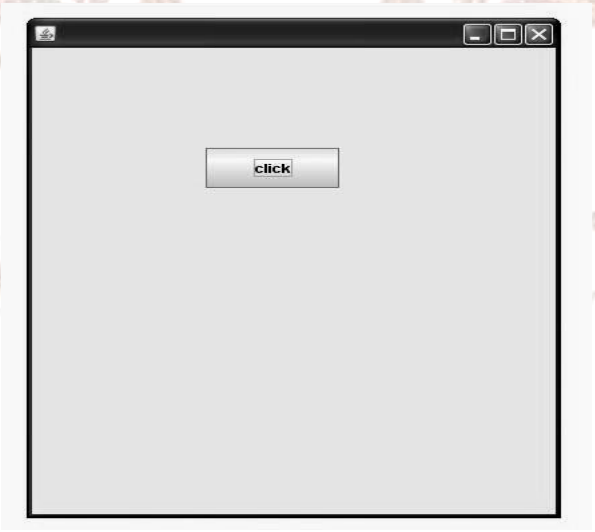


Fig 10.2: Output of FirstSwingExample.java

3.3 Swing Components

A graphical user interface is composed of individual building blocks such as push buttons, scrollbars, and pull-down menus. Some programmers know these individual building blocks as controls, while others call them widgets. In Java, they are typically called components because they all inherit from the base class java.awt.Component.

When you are describing a GUI toolkit, one of the most important characteristics is the list of components it supports. Table 10.2 lists the heavyweight components provided by AWT, where heavyweight refers to components that are layered on top of native GUI components. The components listed are all classes in the java.awt package. One of the curious features of the AWT is that pull-down and pop-up menus, and the items contained within those menus, are not technically components. Instead of inheriting from Component, they inherit from java.awt. MenuComponent. Nevertheless, the various menu component classes are used in very much the same way that true components are.

Table 10.1 lists the components provided by Swing. By convention, the names of these components all begin with the letter J. You'll notice that except for this J prefix, many Swing components have the same names as AWT components. These are designed to be replacements for the corresponding AWT components. For example, the lightweight Swing components JButton and JTextField replace the heavyweight AWT components Button and TextField. In addition, Swing defines a number of components, some quite powerful, that are simply not available in AWT.

Swing components are all part of the javax.swing package. Despite the javax package prefix, Swing is a core part of the Java platform, not a standard extension. Swing can be used as an extension to Java 1.1, however. All Swing components inherit from the javax.swing. JComponent class. JComponent itself inherits from the java.awt.Component class, which means that all Swing components are also AWT components. Unlike most AWT components, however, Swing components do not have a native "peer" object and are therefore "lightweight" components, at least when compared to the AWT components they replace. Finally, note that menus and menu components are no different than any other type of component in Swing; they do not form a distinct class hierarchy as they do in AWT.

Table 10.1: GUI Components of Swing

Component Name	Description
JButton	A push button that can display text, images, or both.
JCheckBox	A toggle button for displaying choices that are not mutually exclusive.
CheckBox-MenuItem	A checkbox designed for use in menus.
JColorChooser	A complex, customizable component that allows the userto select a color from one or more color spaces. Used in conjunction with the javax.swing.color chooser package.
JComboBox	A combination of a text entry field and a drop-down list of choices. The user can type a selection or choose one fromthe list.
JComponent	The root of the Swing component hierarchy. Adds Swing- specific features such as tooltips and support for double- buffering.
JEditorPane	A powerful text editor, customizable via an EditorKit object. Predefined editor kits exist for displaying and editing HTML- and RTF-format text.
JFileChooser	A complex component that allows the user to select a file or directory. Supports filtering and optional file previews. Used in conjunction with the javax.swing.file chooser package.
JLabel	A simple component that displays text, an image, or both. Does not respond to input.
JList	A component that displays a selectable list of choices. Thechoices are usually strings or images, but arbitrary objects may also be displayed.
JMenu	A pull-down menu in a JMenuBar or a submenu withinanother menu.
JMenuBar	A component that displays a set of pull-down menus.
JMenuItem	A selectable item within a menu.
JOptionPane	A complex component suitable for displaying simple dialog boxes. Defines useful static methods for displaying common dialog types.
JPasswordField	A text input field for sensitive data, such as passwords. For security, does not display the text as it is typed.
JPopupMenu	A window that pops up to display a menu. Used by JMenu and for standalone pop-up menus.
JProgressBar	A component that displays the progress of a time- consuming operation.
JRadioButton	A toggle button for displaying mutually exclusive choices.
JRadioButtonMenuI tem	A radio button for use in menus.
JScrollBar	A horizontal or vertical scrollbar.
JSeparator	A simple component that draws a horizontal or vertical line. Used to visually divide complex interfaces into sections.
JSlider	A component that simulates a slider control like those found on stereo equalizers. Allows the user to select a numeric value by dragging a knob. Can display tick marks and labels.
JTable	A complex and powerful component for displaying tables and editing their contents. Typically used to display strings but may be customized to display any type of data. Used in conjunction with the

	javax.swing.table package.
JTextArea	A component for displaying and editing multiple lines of plain text. Based on JTextComponent.
JTextComponent	The root component of a powerful and highly customizable text display and editing system. Part of the javax.swing.textpackage.
JTextField	A component for the display, input, and editing of a single line of plain text. Based on JTextComponent.
JTextPane	A subclass of JEditorPane for displaying and editing formatted text that is not in HTML or RTF format. Suitable for adding simple word processing functionality to an application.
JToggleButton	The parent component of both JCheckBox and JRadio Button.
JToolBar	A component that displays a set of user-selectable tools or actions.
JToolTip	A lightweight pop-up window that displays simple documentation or tips when the mouse pointer lingers over a component.
JTree	A powerful component for the display of tree-structured data. Data values are typically strings, but the component can be customized to display any kind of data. Used in conjunction with the javax.swing.tree package.

3.4 Event Handling

Event-handling is essential to GUI programming. In event handling, the program waits for the user to perform some action. The user controls the sequence of operations that the application executes through a GUI. This approach is also called **event- driven programming.**

An event comprises of three components:

- **Event Object –** When the user interacts with the application by pressing a key or clicking a mouse button, an event is generated. The operating system traps this event and the data associated with it, for example, the time at which the event occurred, the event type (like a ***keypress*** or a ***mouseclick***). This data is then passed on to the application to which the event belongs.

In Java, events are represented by objects that describe the events themselves. Java has a number of classes that describe and handle different categories of event. For example, if you click on a button, an ***ActionEvent*** object is generated. The object of the ***ActionEvent*** class contains information about the event.

- **Event Source –** An event source is an object that generates an event. This is typically an object of the Component class that was described earlier. Event can be generated by objects that do not belong to the Component class also. For example, Windows , Interrupts etc
- **Event-handler** – An event-handler is a method that understands the event and processes it. The event-handler method is invoked whenever an event occurs. The source that generated the event is passed as an input to this method.

3.4.1 Event Classes

The ***EventObject*** class is at the top of the event class hierarchy. It belongs to the ***java.util*** package. Most other event classes are present in the ***java.awt.event*** package. The hierarchy of the event classes is shown below. The ***java.util.EventObject*** and the ***java.awt.AWTEvent*** classes do not belong to the ***java.awt.event*** package. The hierarchy of the JDK event classes is given in the figure 8.1.

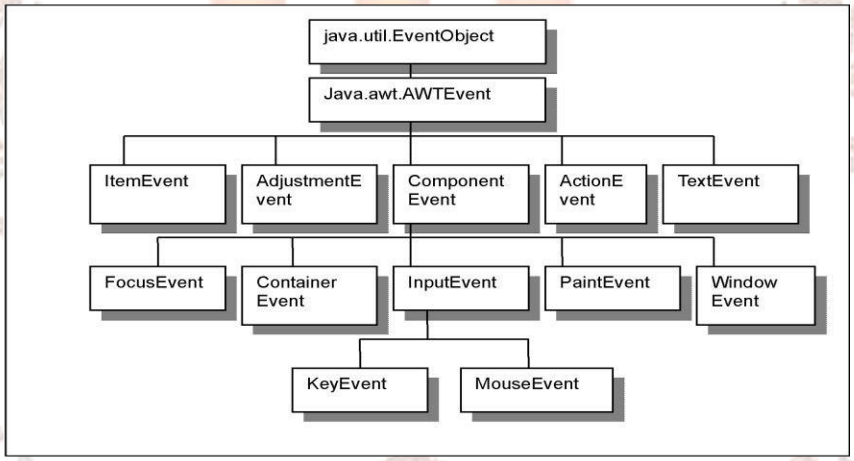


Fig 8.1: Hierarchy of the JDK event classes

The ***getSource()*** method of the ***EventObject*** class returns the object that initiated the event. The ***getID()*** method returns the event ID that represents the nature of the event. For example, if a mouse event occurs, you can find out whether the event was a click, a drag, a move, a press, or a release from the event object.

Let us see when various events are generated:

- An **ActionEvent** object is generated when a component is activated.e.g. button click.

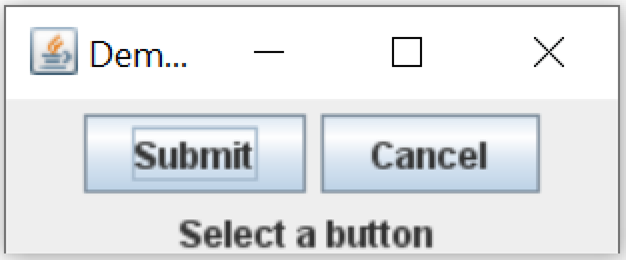
- An **AdjustmentEvent** object is generated when scrollbars and other adjustable elements are used.
- A **ContainerEvent** object is generated when components are added to or removed from a container.
- A **FocusEvent** object is generated when a component receives focus for input.
- An **ItemEvent** object is generated when an item from a list, a choice, or a check box is selected.
- A **KeyEvent** object is generated when a key on the keyboard is pressed.
- A **MouseEvent** object is generated when the mouse is used.
- A **PaintEvent** object is generated when a component is painted.
- A **TextEvent** object is generated when the text of a text component is modified.

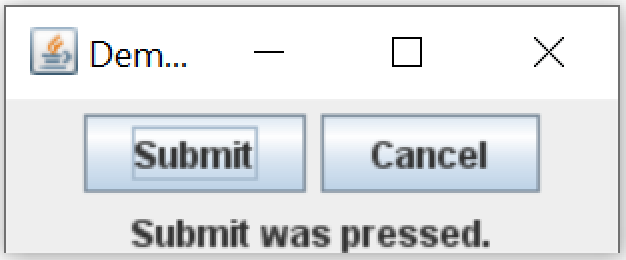
3.4.2 Event Listener

An object delegates the task of handling an event to the event listener. When an event occurs, an event object of the appropriate type (as given above) is created. This object is passed to the listener. A listener must implement the interface that has the method for event-handling. A component can have multiple listeners. A listener can be removed using the ***removeActionListener()*** method. In the figure 8.2 an example has been given showing ActionListener implementation.

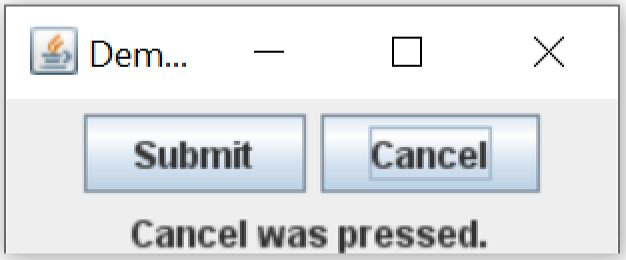
```
public class EventHandling extends JFrame {
    EventHandling() {
            // Create a frame
                  JFrame frame = new JFrame("Demo Button Events");
            frame.setLayout(new FlowLayout());
      JButton jSubmit = new JButton("Submit");
      JButton jCancel = new JButton("Cancel");
      JLabel jLabel = new JLabel("Select a button ");
      jSubmit.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                  jLabel.setText("Submit was pressed.");
            }
      });
      jCancel.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                  jLabel.setText("Cancel was pressed.");
            }
      });
      frame.add(jSubmit);
      frame.add(jCancel);
      frame.add(jLabel);
}

public static void main(String args[]) {
      // Create the frame on the event dispatching thread.
      SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                  new EventHandling();
            }
      });
}
```





Submit was pressed.



Cancel was pressed.

SELF ASSESSMENT QUESTIONS – 1

1. When the user interacts with the application by pressing a key or clicking a mouse button, an__________________is generated.
2. The_________class is at the top of the event class hierarchy.

3.4.3 Event-Handling

When an event occurs, it is sent to the component from where the event originated. The component registers a listener, which contains event- handler. Event-handler receives and process events. The Event handling process has been given in the figure 8.3.

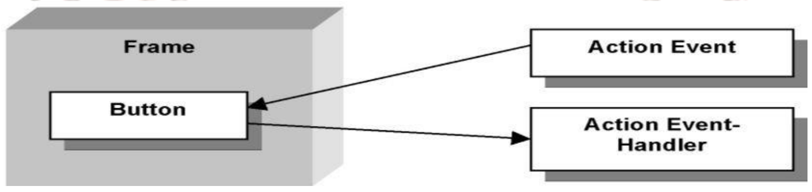


Fig 10.3: Event handling in Java

Every event has a corresponding listener interface that specifies the methods that are required to handle the event. Event objects report to the registered listeners. To enable a component to handle events, you must register an appropriate listener for the component. The example of Frame is given in the figure 8.4.

```
class ButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent evt) {
                Button source = (Button) evt.getSource();
                source.setLabel("Button clicked");
}
```

How does the above application work?

- ❖ A listener object is created.
- ❖ The addActionListener() method registers the listener object for the button.
- ❖ The application waits for the user to interact with it.
- ❖ When the user clicks on the button:
  - The ***ActionEvent*** event is generated.
  - An ***ActionEvent*** object is created and is delegated to the registered listener object for processing.
  - The listener object contains the ***actionPerformed()*** method, which processes the ***ActionEvent.***
  - In the ***actionPerformed()*** method, the reference to the event source is retrieved using the ***getSource()*** method.
  - Finally, the label of the button is changed using the ***setLabel()***

method.

SELF ASSESSMENT QUESTIONS – 2

3. When the user clicks on the button, the_________event is generated.
4. An________________event is generated by movement of a mouse.

3.4.4 Adapter Classes

Implementing the listener interface implies that we have to provide an implementation for all the methods in the interface. Alternately, the Java programming language provides adapter classes that implement the corresponding listener interfaces containing more than one method. The methods in these classes are empty. The listener class that you define can extend the Adapter class and override only those methods that we are interested in.

Event Category	**Interface Name**	**AdapterName**	**Method**
Window	Window-Listener	Window-Adapter	void windowClosing (WindowEvent e) void windowOpening (WindowEvent e) void windowActivated (WindowEvent e) void windowDeactivated (WindowEvent e) void windowClosed (WindowEvent e) void windowIconified (WindowEvent e) void windowDeiconified (WindowEvent e)
Action	ActionListenr		void actionPerformed (ActionEvent e)
Item	ItemListener		void itemStateChanged (ItemEvent e)
Mouse Motion	Mouse-Motion Listener	Mouse-Motion Adapter	void mouseDragged (MouseEvent e)void mouseMoved (MouseEvent e)
Mouse Button	Mouse-Listener	Mouse-Adapter	void mousePressed (MouseEvent e) void mouseReleased (MouseEvent e) void mouseEntered (MouseEvent e) void mouseExited (MouseEvent e) void mouseClicked (MouseEvent e)
Key	KeyListener	Key- Adapter	void keyPressed (KeyEvent e) void keyReleased (KeyEvent e) void keyTyped (KeyEvent e)
Focus	Focus-Listener		void focusGained (FocusEvent e) void focusLost (FocusEvent e)
Component	Component-Listener	Compo-nent Adapter	Void componentMoved (ComponentEvent e) void componentResized (ComponentEvent e) void componentHidden (ComponentEvent e) void componentShown (ComponentEvent e)

4. 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 figure 10.3 shows the architecture of JavaFX API. Here you can see the components that support JavaFX API.

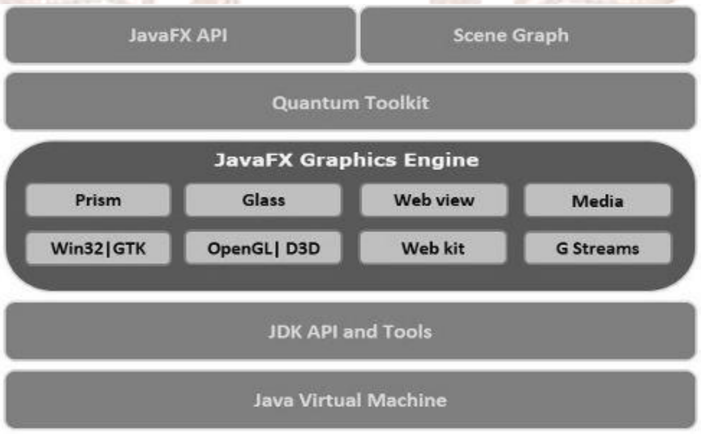


Fig 10.3: Architecture of JavaFX API

4.1 Concepts

4.1.1 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 in the figure 10.4.

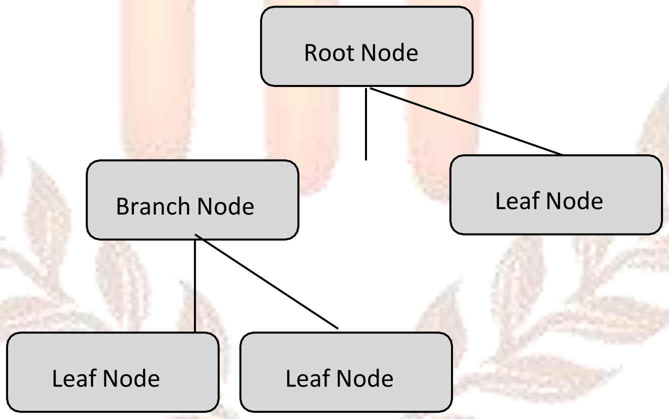


Fig 10.4: Java Scene Graph

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.

4.1.2 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.

4.1.3 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.

4.1.4 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.

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.

In general, a JavaFX application will have three major components namely Stage, Scene and Nodes as shown in the figure 10.5.

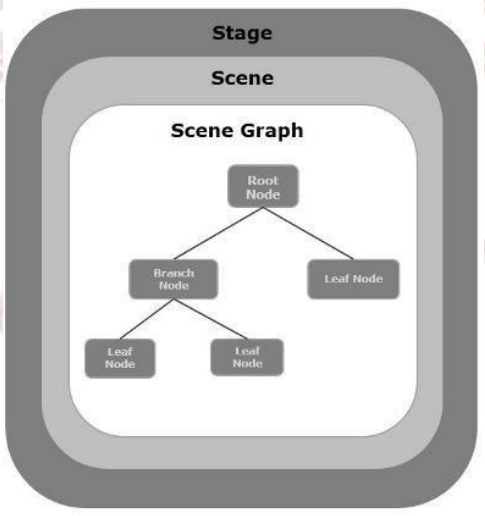


Fig 10.5: JavaFX Application Structure

4.2 Creating a JavaFX FXML Application

A helpful structural tool with JavaFX is to specify JavaFX scene graph nodes with FXML. FXML is an XML markup language that fits nicely with the hierarchical nature of a scene graph. FXML helps to visualize scene graph structures and lends itself to easier modification that can otherwise be tedious with Java code.

FXML typically requires three files: the program's main Java file, the FXML file, and a Java controller class for the FXML file. The main Java class uses an FXML Loader to create the Stage and Scene. The FXML Loader reads the FXML file and builds the scene graph. The controller class provides JavaFX node initialization code and accesses the scene graph programmatically to create dynamic content or handle events.

```
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class MyRectangleFXApp extends Application {

       @Override
       public void start ( Stage stage) throws Exception {
              Parent root = FXMLLoader
                          .load(getClass()
                          .getResource("myRectangleFX.fxml"));
              Scene scene = new Scene (root, Color.LIGHTBLUE);
              stage.setScene(scene);
              stage.show();
       }

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

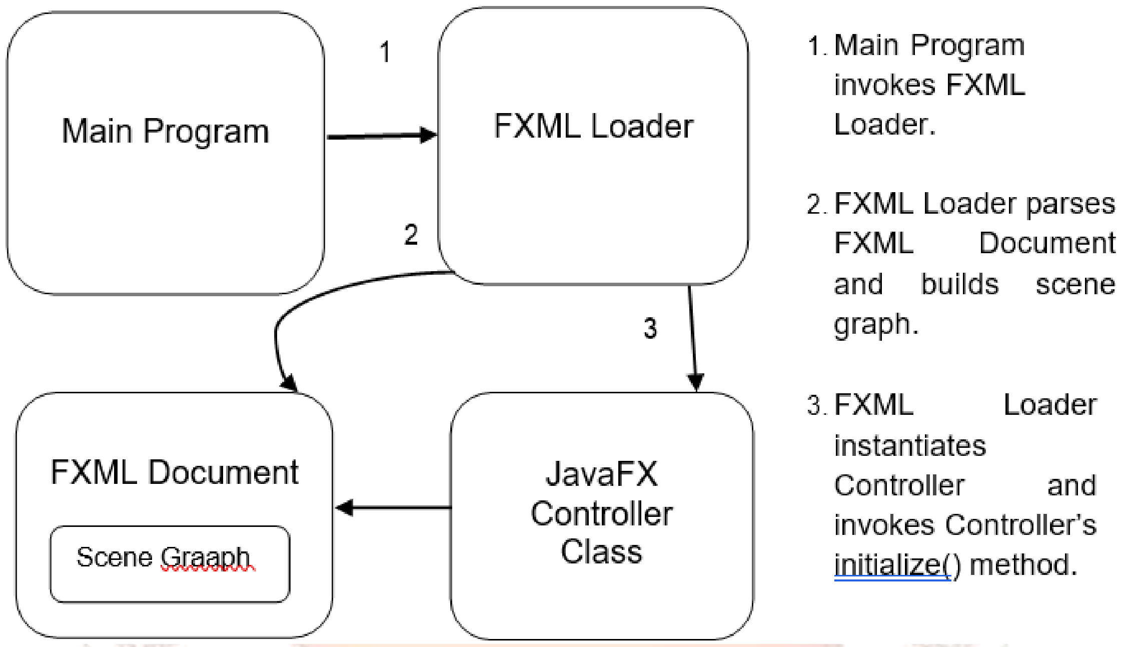

Main Program
1
FXML Loader
2
3
FXML Document
Scene Graaph
JavaFX Controller Class
1. Main Program invokes FXML Loader.
2. FXML Loader parses FXML Document and builds scene graph.
3. FXML Loader instantiates Controller and invokes Controller's initialize() method.

Fig 10.6: Structure of a JavaFX FXML Application

Figure 10.6 shows the structure of a JavaFX FXML Application. Execution begins with the main program, which invokes the FXML Loader. The FXML Loader parses the FXML document, instantiates the objects, and builds the scene graph. After building the scene graph, the FXML Loader instantiates the controller class and invokes the controller's initialize() method.

Now let's look at the FXML markup for this application, as shown in the figure 10.10 Each FXML file is associated with a controller class, specified with the fx:controller attribute (marked in bold). With XML markup, you see that the structure of the FXML matches the hierarchical layout depicted in Figure 3.7 on page 92 (that is, starting with the root node, StackPane). The StackPane is the top node and its children are the Rectangle and Text nodes. The Rectangle's properties are configured with property names and values that are converted to the correct types. The style property matches element -fx-fill we showed you earlier. Both the Rectangle and Text elements have their effect properties configured. The Rectangle has a drop shadow and the Text element has a reflection effect.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.StackPane?>
<?import javafx.scene.* ?>
<?import javafx.scene.control.* ?>
<?import javafx.scene.layout.* ?>
<?import javafx.scene.shape.* ?>
<?import javafx.scene.text.* ?>
<?import javafx.scene.effect.* ?>

<StackPane prefHeight="400.0" prefWidth="600.0"
      xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml/1">
      <children>
            <Rectangle fx:id="rectangle" width="200" height="100"
                  arcWidth="30" arcHeight="30"
                  style="-fx-fill: linear-gradient(from 25px 25px to 100px 100px,
#dc143c, #32cd32);">
                  <effect>
                        <DropShadow color="GRAY" offsetX="5.0" offsetY="5.0" />
                  </effect>
            </Rectangle>
            <Text text="My Rectangle" >
                  <effect>
                        <Reflection />
                  </effect>
                  <font>
                        <Font name="Verdana Bold" size="18.0" />
                  </font>
            </Text>
      </children>

</StackPane>
```

Fig 10.7: FXML markup

To access FXML elements from the controller class, give them an fx-id tag. Here, the Rectangle element has been assigned fx:id="rectangle". This references a class variable that you declare in the controller class.

Now let's show you the controller class. Figure 10.8 displays the source for MyRectangleFXController.java.

```java
import java.net.URL;
import java.util.ResourceBundle;

import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;

public class MyRectangleController implements Initializable{

	@FXML
	private Rectangle rectangle;

	@Override
	public void initialize(URL arg0, ResourceBundle arg1) {
		// TODO Auto-generated method stub
		rectangle.setStrokeWidth(5.0);
		rectangle.setStroke(Color.GOLDENROD);

	}

}
```

Fig 10.7: MyRectangleFXController

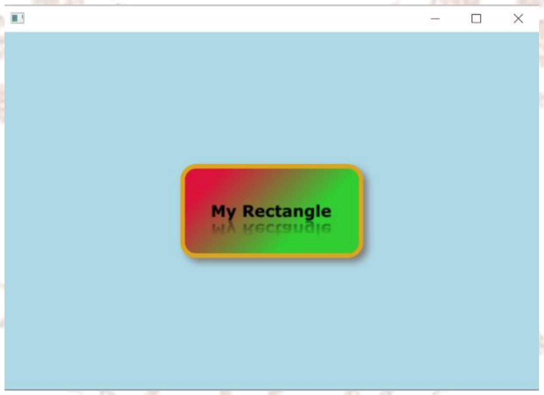


Figure 10.8 shows MyRectangleFXApp running with the Rectangle's stroke and strokeWidth properties configured.

Fig 10.8: Rectangle's customized stroke and strokeWidth properties

4.3 Using the CSS Files

You can specify a CSS file either directly in the FXML file or in the main program. To specify a CSS file in the main program, add a call to scene.getStylesheets(), as shown in the figure 10.9.

```
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class MyRectangleFXApp extends Application {

       @Override
       public void start ( Stage stage) throws Exception {
              Parent root = FXMLLoader
                                          .load(getClass()
                                          .getResource("myRectangleFX.fxml"));
              Scene scene = new Scene (root, Color.LIGHTBLUE);
              scene.getStylesheets().add("MyCSS.css");
              stage.setScene(scene);
              stage.show();
       }

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

Fig 10.9: Adding a CSS Style Sheet in the Main Program

It's also possible to specify the style sheet in FXML, as shown in the figure 10.10. Note that you must include java.net.* to define element URL.

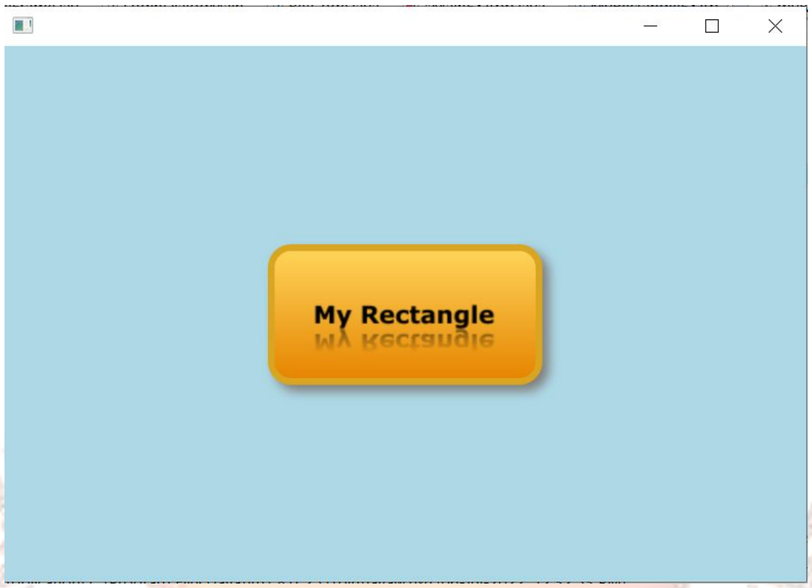
```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.StackPane?>
<?import javafx.scene.* ?>
<?import javafx.scene.control.* ?>
<?import javafx.scene.layout.* ?>
<?import javafx.scene.shape.* ?>
<?import javafx.scene.text.* ?>
<?import javafx.scene.effect.* ?>

<StackPane prefHeight="400.0" prefWidth="600.0"
      xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml/1"
      fx:controller = "MyRectangleController" stylesheets="@MyCSS.css">
      <children>
            <Rectangle fx:id="rectangle" width="200" height="100"
                  arcWidth="30" arcHeight="30">
                  <effect>
                        <DropShadow color="GRAY" offsetX="5.0" offsetY="5.0" />
                  </effect>
            </Rectangle>
            <Text text="My Rectangle" >
                  <effect>
                        <Reflection />
                  </effect>
                  <font>
                        <Font name="Verdana Bold" size="18.0" />
                  </font>
            </Text>
      </children>

</StackPane>
```

MyRectangleFX.fxml – Adding a Style Sheet



Take a look at the Rectangle element in Program 10.6. It's much shorter now since the FXML no longer includes the style or effect property values. The FXML does, however, contain a property id value. This id attribute identifies the node for the CSS style definition.

We've also defined an event handler for a mouse clicked event in the Rectangle FXML (#handleMouseClick). The onMouseClicked attribute lets you wire an event handler with an FXML component. The mouse clicked event handler is invoked when the user clicks inside the Rectangle.

Finally, as shown in Program 10.6, we modified the StackPane to include element fx:id="stackpane" so we can refer to the StackPane in the controller code.

Figure 10.11 shows the CSS file MyCSS.css, which defines a style specifically for the component with id "myrectangle."

```
#rectangle {
-fx-fill:
linear-gradient(#ffd65b, #e68400),
linear-gradient(#ffef84,#f2ba44),
linear-gradient(#ffea6a, #efaa22),
linear-gradient(#ffe657 0%, #f8c202 50%, #eea10b 100%),
linear-gradient(from 0% 0% to 15% 50%, rgba(255,255,255,0.9), rgba(255,255,255,0));
-fx-effect: dropshadow( three-pass-box, gray, 10, 0 , 5.0, 5.0);
}
```

Fig 10.11: MyCSS.css

4.4 Layout Pane

There are two types of layouts to arrange nodes in a scene graph:

- Static layout
- Dynamic layout

In a static layout, the position and size of nodes are calculated once, and they stay the same as the window is resized. The user interface looks good when the window has the size for which the nodes were originally laid out.

In a dynamic layout, nodes in a scene graph are laid out every time a user action necessitates a change in their position, size, or both. Typically, changing the position or size of one node affects the position and size of all other nodes in the scene graph. The dynamic layout forces the recomputation of the position and size of some or all nodes as the window is resized.

Both static and dynamic layouts have advantages and disadvantages. A static layout gives developers full control on the design of the user interface. It lets you make use of the available space as you see fit. A dynamic layout requires more programming work, and the logic is much more involved. Typically, programming languages supporting GUI: for example, JavaFX, supports dynamic layouts through libraries.

Libraries solve most of the use-cases for dynamic layouts. If they do not meet your needs, you must do the hard work to roll out your own dynamic layout.

A layout pane is a node that contains other nodes, which are known as its children (or child nodes). The responsibility of a layout pane is to lay out its children, whenever needed. A layout pane is also known as a container or a layout container.

A layout pane has a layout policy that controls how the layout pane lays out its children. For example, a layout pane may lay out its children horizontally, vertically, or in any other fashion.

JavaFX contains several layout-related classes, which are the topic of discussion in this chapter. A layout pane performs two things:

- It computes the position (the x and y coordinates) of the node within its parent.
- It computes the size (the width and height) of the node.

For a 3D node, a layout pane also computes the z coordinate of the position and the depth of the size. The layout policy of a container is a set of rules to compute the position and size of its children. A node has three sizes: preferred size, minimum size, and maximum size. Most of the containers attempt to give its children their preferred size. The actual (or current) size of a node may be different from its preferred size. The current size of a node depends on the size of the window, the layout policy of the container, and the expanding and shrinking policy for the node, etc.

Layout Pane Classes

JavaFX contains several container classes. Figure 10.16 shows a class diagram for the container classes. A container class is a subclass, direct or indirect, of the Parent class.

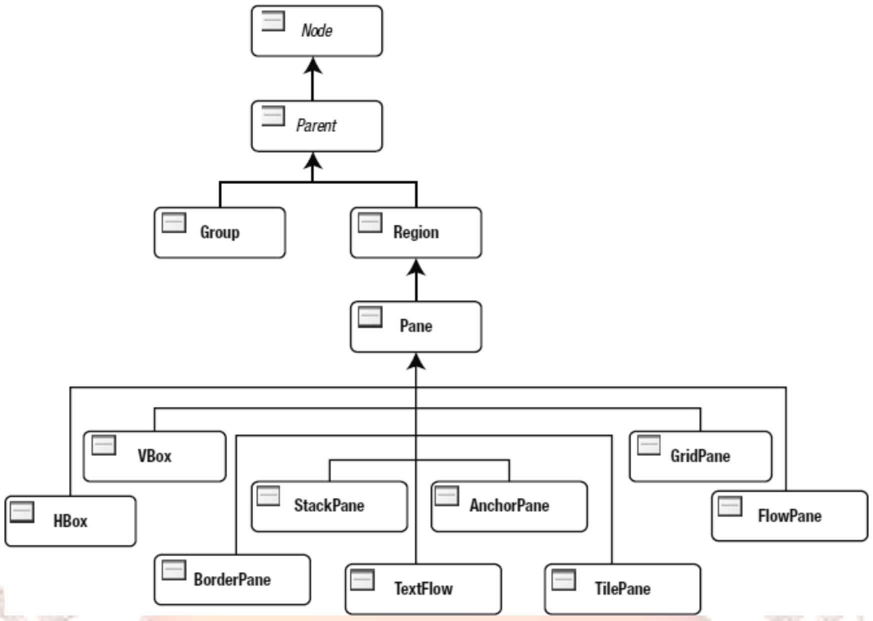


Fig 10.12: class diagram for the container classes

Table 10.2 has brief descriptions of the container classes.

Table 10.2: List of Container Classes

Container Class	**Description**
Group	A Group applies effects and transformations collectively to all itschildren.
Pane	It is used for absolute positioning of its children.
HBox	It arranges its children horizontally in a single row.
VBox	It arranges its children vertically in a single column.
FlowPane	It arranges its children horizontally or vertically in rows or columns. Ifthey do not fit in a single row or column, they are wrapped at the specified width or height.
BorderPane	It divides its layout area in the top, right, bottom, left, and center regions and places each of its children in one of the five regions.
StackPane	It arranges its children in a back-to-front stack.
TitlePane	It arranges its children in a grid of uniformly sized cells.
GridPane	It arranges its children in a grid of variable sized cells.
AnchorPane	It arranges its children by anchoring their edges to the edges of thelayout area.
TextFlow	It lays out rich text whose content may consist of several Text nodes.

SELF ASSESSMENT QUESTIONS – 3

5. contains classes to add transition based animationssuch as fill, fade, rotate, scale and translation, to the JavaFX nodes.
6. Java Foundation Classes are superset that contains AWT andcompletely written in__________________________.
7. Swing components are all part of the____________package.
8. A scene represents the_____________of a JavaFX application.

5. SUMMARY

- Java Foundation Classes are a set of Graphical User Interface. Java Foundation Classes are superset that contains AWT and completely written in java. Unlike AWT, Java Swing provides platform-independent and lightweight components.
- Events can be handled to track the activity on the components, window etc .
- JavaFX provides a complete API with a rich set of classes and interfaces to build GUI applications with rich graphics.
- 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 stage (a window) contains all the objects of a JavaFX application. It is represented by Stage class of the package javafx.stage.
- A scene represents the physical contents of a JavaFX application. It contains all the contents of a scene graph.
- FXML is an XML markup language that fits nicely with the hierarchical nature of a scene graph. FXML helps to visualize scene graph structures and lends itself to easier modification that can otherwise be tedious with Java code.
- There are two types of layouts to arrange nodes in a scene graph: Static layout and Dynamic layout
- In a static layout, the position and size of nodes are calculated once, and they stay the same as the window is resized. The user interface looks good when the window has the size for which the nodes were originally laid out. In a dynamic layout, nodes in a scene

graph are laid out every time a user action necessitates a change in their position, size, or both.

6. TERMINAL QUESTIONS

1. What are the important packages of Java Swing?
2. What are the important packages of JavaFX?
3. What are the important steps required to create a JavaFX FXML Application?

7. ANSWERS

Self Assessment Questions

1. event
2. EventObject
3. Action
4. MouseEvent
5. javafx.animation
6. Java
7. javax.swing
8. physical contents

Terminal Questions

1. Swing's packages, and their contents, are:

   The Accessibility package, The Swing component package (javax.swingapi, The basic package (javax.swing.plaf.basic), The beaninfo package (javax.swing.beaninfo), The border package (javax. swing.borderapi), The event package (javax.swing.eventapi), The multi package (javax.swing.plaf.multi), The pluggable L&F package (javax. swing.plaf),The table package (javax.swing.tableapi), The text package (javax.swing.textapi), The HTML package (javax.swing.text.htmlapi), The RTF package (javax.swing.text.rtfapi), The tree package (javax. swing.treeapi), The undo package (javax.swing.undoapi): (Refer Section 3)

2. The important packages of this API are – javafx.animation, javafx. application, javafx.css, javafx.event,, javafx.geometry, javafx.stage, javafx.scene (Refer Section 6)

3. FXML typically requires three files: the program's main Java file, the FXML file, and a Java controller class for the FXML file. The main Java class uses an FXML Loader to create the Stage and Scene. The FXML Loader reads the FXML file and builds the scene graph. The controller class provides JavaFX node initialization code and accesses the scene graph programmatically to create dynamic content or handle events. (Refer Section 6.5)