UNIT V

GUI PROGRAMMING

Introduction to Swing – Model-View-Controller design pattern – layout management – Swing Components -Introduction to JavaFX - JavaFX components

Swing:

Swing is a Java Foundation Classes [JFC] library and an extension of the Abstract Window Toolkit [AWT]. Java Swing offers much-improved functionality over AWT, new components, expanded components features, and excellent event handling with drag-and-drop support.

Introduction to Swing:

Swing has about four times the number of User Interface [UI] components as AWT and is part of the standard Java distribution. By today's application GUI requirements, AWT is a limited implementation, not quite capable of providing the components required for developing complex GUIs required in modern commercial applications. The AWT component set has quite a few bugs and does take up a lot of system resources when compared to equivalent Swing resources. Netscape introduced its Internet Foundation Classes [IFC] library for use with Java. Its Classes became very popular with programmers creating GUI's for commercial applications.

- Swing is a Set of API (API- Set of Classes and Interfaces)
- Swing is Provided to Design Graphical User Interfaces

- Swing is an Extension library to the AWT (Abstract Window Toolkit)
- Includes New and improved Components that have been enhancing the looks and Functionality of GUIs'
- Swing can be used to build (Develop) The Standalone swing GUI Apps as Servlets and Applets
- It Employs model/view design architecture.
- Swing is more portable and more flexible than AWT, the Swing is built on top of the AWT.
- Swing is Entirely written in Java.
- Java Swing Components are Platform-independent, and The Swing Components are lightweight.
- Swing Supports a Pluggable look and feel and Swing provides more powerful components.
- such as tables, lists, Scroll panes, Colour chooser, tabbed pane, etc.
- Further Swing Follows MVC.

Difference between Java Swing and Java AWT

There are certain points from which Java Swing is different than Java AWT as mentioned below:

Java AWT	**Java Swing**
AWT components are platform-dependent.	Java swing components are platform-independent.
AWT components are heavyweight.	The components of Java Swing are lightweight.
AWT doesn't support pluggable look and feel.	Swing supports pluggable look and feel.

AWT provides less components than Swing.	Swing provides more powerful components such as tables, lists, scroll panes, color chooser, tabbed pane etc.
AWT components require java.awt package.	Swing components requires javax.swing package.

JFC:

The Java Foundation Classes (JFC) are a set of GUI components which simplify the development of desktop applications.

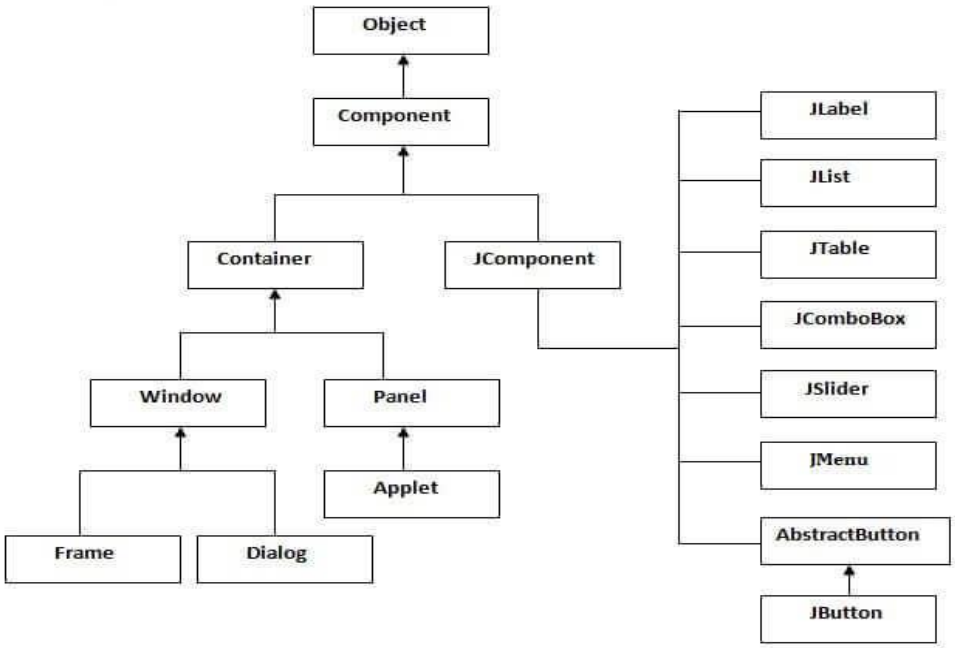
Swing Components:

The methods of Component class are widely used in java swing that are given below.

Method	Description
public void add(Component c)	add a component on another component.
public void setSize(int width,int height)	sets size of the component.
public void setLayout(LayoutManager m)	sets the layout manager for the component.
public void setVisible(boolean b)	sets the visibility of the component. It is by default false.

Hierarchy of Java Swing classes:

The hierarchy of java swing API is given below.



Java Swing

Examples

There are two ways to create a frame:

- By creating the object of Frame class (association)
- By extending Frame class (inheritance)

We can write the code of swing inside the main(), constructor or any other method.

Example 1:

```
import javax.swing.*;
public class FirstSwingExample {
public static void main(String[] args) {
JFrame f=new JFrame();//creating instance of JFrame
JButton b=new JButton("click");//creating instance of JButton
b.setBounds(130,100,100, 40);//x axis, y axis, width, height
f.add(b);//adding button in JFrame
```

```
f.setSize(400,500);//400 width and 500 height
fsetLayout(null);//using no layout managers
f.setVisible(true);//making the frame visible
}
}
```

Example 2:

```
import javax.swing.*;
public class Simple {
JFrame f;
Simple(){
f=new JFrame();//creating instance of JFrame
JButton b=new JButton("click");//creating instance of JButton
b.setBounds(130,100,100, 40);
f.add(b);//adding button in JFrame
f.setSize(400,500);//400 width and 500 height
f.setLayout(null);//using no layout managers  f.setVisible(true);//mak
ing the frame visible
}
public static void main(String[] args) {
new Simple();
}
    }
```

Model-View-Controller (MVC):

The Model-View-Controller (MVC) is a well-known design pattern in the web development field. It is way to organize our code. It specifies that a program or application shall consist of data model, presentation

information and control information. The MVC pattern needs all these components to be separated as different objects.

In this section, we will discuss the MVC Architecture in Java, along with its advantages and disadvantages and examples to understand the implementation of MVC in Java.

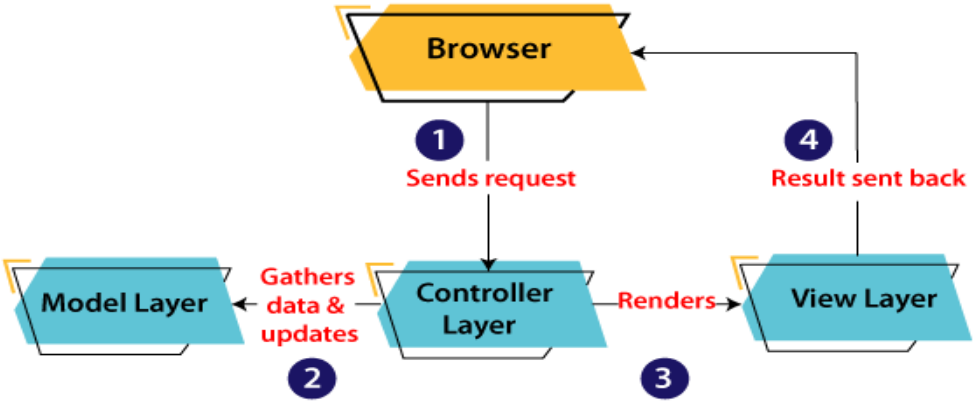
What is MVC architecture in Java?

The model designs based on the MVC architecture follow MVC design pattern. The application logic is separated from the user interface while designing the software using model designs.

The MVC pattern architecture consists of three layers:

- **Model:** It represents the business layer of application. It is an object to carry the data that can also contain the logic to update controller if data is changed.
- **View:** It represents the presentation layer of application. It is used to visualize the data that the model contains.
- **Controller:** It works on both the model and view. It is used to manage the flow of application, i.e. data flow in the model object and to update the view whenever data is changed.

In Java Programming, the Model contains the simple Java classes, the View used to display the data and the Controller contains the servlets. Due to this separation the user requests are processed as follows:



1. A client (browser) sends a request to the controller on the server side, for a page.
2. The controller then calls the model. It gathers the requested data.
3. Then the controller transfers the data retrieved to the view layer.
4. Now the result is sent back to the browser (client) by the view.

Advantages of MVC Architecture:

The advantages of MVC architecture are as follows,

- MVC has the feature of scalability that in turn helps the growth of application.
- The components are easy to maintain because there is less dependency.
- A model can be reused by multiple views that provides reusability of code.
- The developers can work with the three layers (Model, View, and Controller) simultaneously.
- Using MVC, the application becomes more understandable.
- Using MVC, each layer is maintained separately therefore we do not require to deal with massive code.
- The extending and testing of application is easier.

Implementation of MVC using Java

To implement MVC pattern in Java, we are required to create the following three classes.

- **Employee Class**, will act as model layer.
- **EmployeeView Class**, will act as a view layer.
- **EmployeeContoller Class**, will act a controller layer.

MVC Architecture Layers:

Model Layer:

The Model in the MVC design pattern acts as a data layer for the application. It represents the business logic for application and also the state of application. The model object fetches and store the model state in the database. Using the model layer, rules are applied to the data that represents the concepts of application.

Let's consider the following code snippet that creates a which is also the first step to implement MVC pattern.

Example:

```
// class that represents model
public class Employee {
// declaring the variables
private String EmployeeName;
private String EmployeeId;
private String EmployeeDepartment;
// defining getter and setter methods
public String getId() {
```

```
return EmployeeId;
}
public void setId(String id) {
this.EmployeeId = id;
}
public String getName() {
return EmployeeName;
}
public void setName(String name) {
this.EmployeeName = name;
}
public String getDepartment() {
return EmployeeDepartment;
}
public void setDepartment(String Department) {
this.EmployeeDepartment = Department;
}
}
```

View Layer:

As the name depicts, view represents the visualization of data received from the model. The view layer consists of output of application or user interface. It sends the requested data to the client, that is fetched from model layer by controller.

Let's take an example where we create a view using the EmployeeView class.

Example:

```
// class which represents the view
public class EmployeeView {
// method to display the Employee details
public void printEmployeeDetails (String EmployeeName, String Em
ployeeId, String EmployeeDepartment){
System.out.println("Employee Details: ");
System.out.println("Name: " + EmployeeName);
System.out.println("Employee ID: " + EmployeeId);
System.out.println("Employee Department: " + EmployeeDepartment
);
}
}
```

Controller Layer:

The controller layer gets the user requests from the view layer and processes them, with the necessary validations. It acts as an interface between Model and View. The requests are then sent to model for data processing. Once they are processed, the data is sent back to the controller and then displayed on the view.

Let's consider the following code snippet that creates the controller using the EmployeeController class.

Example:

```
// class which represent the controller
public class EmployeeController {
// declaring the variables model and view
```

```
private Employee model;
private EmployeeView view;
// constructor to initialize
public EmployeeController(Employee model, EmployeeView view) {
this.model = model;
this.view = view;
}
// getter and setter methods
public void setEmployeeName(String name){
model.setName(name);
}
public String getEmployeeName(){
return model.getName();
}
public void setEmployeeId(String id){
model.setId(id);
}
public String getEmployeeId(){
return model.getId();
}
public void setEmployeeDepartment(String Department){
model.setDepartment(Department);
}
public String getEmployeeDepartment(){
return model.getDepartment();
```

```
}
// method to update view
public void updateView() {
view.printEmployeeDetails(model.getName(), model.getId(), model.getDepartment());
}
}
```

Main Class Java file:

The following example displays the main file to implement the MVC architecture. Here, we are using the MVC Main class.

```
// main class
public class MVCMain {
public static void main(String[] args) {
// fetching the employee record based on the employee_id from the database
Employee model = retriveEmployeeFromDatabase();
// creating a view to write Employee details on console
EmployeeView view = new EmployeeView();
EmployeeController controller = new EmployeeController(model, view);
controller.updateView();
//updating the model data
controller.setEmployeeName("Nirnay");
System.out.println("\n Employee Details after updating: ");
controller.updateView();
```

```
}
private static Employee retriveEmployeeFromDatabase(){
Employee Employee = new Employee();
Employee.setName("Anu");
Employee.setId("11");
Employee.setDepartment("Salesforce");
return Employee;
}
}
```

The MVC Main class fetches the employee data from the method where we have entered the values. Then it pushes those values in the model. After that, it initializes the view (EmployeeView.java). When view is initialized, the Controller (EmployeeController.java) is invoked and bind it to Employee class and EmployeeView class. At last the updateView() method (method of controller) update the employee details to be printed to the console.

Output:

```
Employee Details:
Name: Anu
Employee ID: 11
Employee Department: Salesforce
Employee Details after updating:
Name: Nirnay
Employee ID: 11
Employee Department: Salesforce
```

Java Layout Management:

The Layout Managers are used to arrange components in a particular manner. The Java Layout Managers facilitates us to control the positioning and size of the components in GUI forms. Layout Manager is an interface that is implemented by all the classes of layout managers.

Java BorderLayout

The BorderLayout is used to arrange the components in five regions: north, south, east, west, and center. Each region (area) may contain one component only. It is the default layout of a frame or window. The BorderLayout provides five constants for each region:

1. **public static final int NORTH**
2. **public static final int SOUTH**
3. **public static final int EAST**
4. **public static final int WEST**
5. **public static final int CENTER**

Constructors of BorderLayout class:

- **BorderLayout():** creates a border layout but with no gaps between the components.
- **BorderLayout(int hgap, int vgap):** creates a border layout with the given horizontal and vertical gaps between the components.

Example 1:

```
import java.awt.*;
import javax.swing.*;
public class Border
{
JFrame f;
Border()
{
f = new JFrame();
// creating buttons
JButton b1 = new JButton("NORTH");; // the button will be labeled a
s NORTH
JButton b2 = new JButton("SOUTH");; // the button will be labeled as
 SOUTH
JButton b3 = new JButton("EAST");; // the button will be labeled as
EAST
JButton b4 = new JButton("WEST");; // the button will be labeled as
WEST
JButton b5 = new JButton("CENTER");; // the button will be labeled
as CENTER
f.add(b1, BorderLayout.NORTH); // b1 will be placed in the North Di
rection
f.add(b2, BorderLayout.SOUTH);  // b2 will be placed in the South Di
rection
f.add(b3, BorderLayout.EAST);  // b2 will be placed in the East Direc
tion
```

```
f.add(b4, BorderLayout.WEST);  // b2 will be placed in the West Dire
ction
f.add(b5, BorderLayout.CENTER);  // b2 will be placed in the Center
f.setSize(300, 300);
f.setVisible(true);
}
public static void main(String[] args) {
new Border();
}
}
```

Example 2:

```
// import statement
import java.awt.*;
import javax.swing.*;
public class BorderLayoutExample
{
JFrame jframe;
// constructor
BorderLayoutExample()
{
// creating a Frame
jframe = new JFrame();
// create buttons
JButton btn1 = new JButton("NORTH");
```

```
JButton btn2 = new JButton("SOUTH");
JButton btn3 = new JButton("EAST");
JButton btn4 = new JButton("WEST");
JButton btn5 = new JButton("CENTER");
// creating an object of the BorderLayout class using
// the parameterized constructor where the horizontal gap is 20
// and vertical gap is 15. The gap will be evident when buttons are pla
ced
// in the frame
jframe.setLayout(new BorderLayout(20, 15));
jframe.add(btn1, BorderLayout.NORTH);
jframe.add(btn2, BorderLayout.SOUTH);
jframe.add(btn3, BorderLayout.EAST);
jframe.add(btn4, BorderLayout.WEST);
jframe.add(btn5, BorderLayout.CENTER);
jframe.setSize(300,300);
jframe.setVisible(true);
}
// main method
public static void main(String argvs[])
{
new BorderLayoutExample();
}
}
```

Java BorderLayout - Without Specifying Region:

The add() method of the JFrame class can work even when we do not specify the region. In such a case, only the latest component added is shown in the frame, and all the components added previously get discarded. The latest component covers the whole area. The following example shows the same.

Example:

```
 // import statements
import java.awt.*;
import javax.swing.*;
public class BorderLayoutWithoutRegionExample
{
JFrame jframe;
// constructor BorderLayoutWithoutRegionExample()
{
jframe = new JFrame();
JButton btn1 = new JButton("NORTH");
JButton btn2 = new JButton("SOUTH");
JButton btn3 = new JButton("EAST");
JButton btn4 = new JButton("WEST");
JButton btn5 = new JButton("CENTER");
// horizontal gap is 7, and the vertical gap is 7
// Since region is not specified, the gaps are of no use
jframe.setLayout(new BorderLayout(7, 7));
// each button covers the whole area
```

```
// however, the btn5 is the latest button
// that is added to the frame; therefore, btn5
// is shown
jframe.add(btn1);
jframe.add(btn2);
jframe.add(btn3);
jframe.add(btn4);
jframe.add(btn5);
jframe.setSize(300,300);
jframe.setVisible(true);
}
// main method public static void main(String argvs[])
{
new BorderLayoutWithoutRegionExample();
}
}
```

Introduction to JavaFX:

JavaFX is a Java library used to develop Desktop applications as well as Rich Internet Applications (RIA). The applications built in JavaFX, can run on multiple platforms including Web, Mobile and Desktops. JavaFX is intended to replace swing in Java applications as a GUI framework. However, It provides more functionalities than swing. Like Swing, JavaFX also provides its own components and doesn't depend upon the operating system. It is lightweight and hardware accelerated. It supports various operating systems including Windows, Linux and Mac OS.

Features of JavaFX:

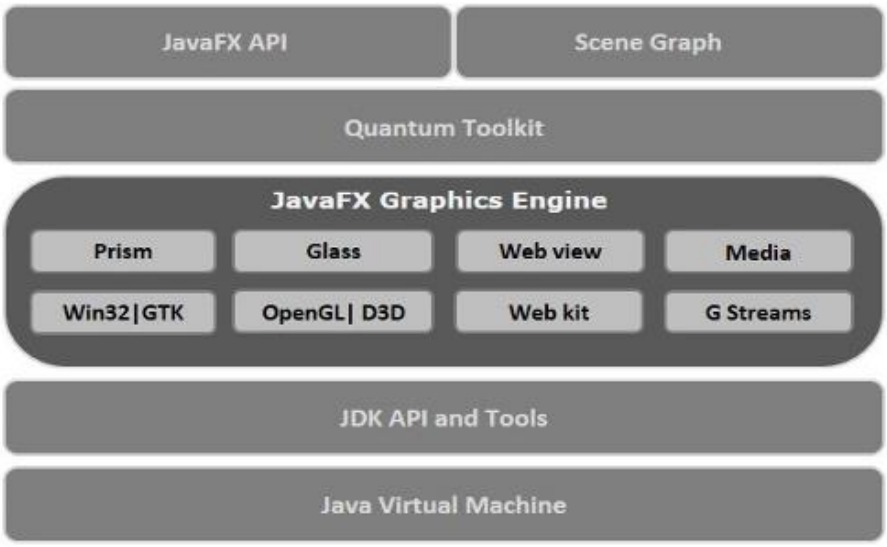
Feature	Description
Java Library	It is a Java library which consists of many classes and interfaces that are written in Java.
FXML	FXML is the XML based Declarative mark up language. The coding can be done in FXML to provide the more enhanced GUI to the user.
Scene Builder	Scene Builder generates FXML mark-up which can be ported to an IDE.
Web view	Web pages can be embedded with JavaFX applications. Web View uses WebKitHTML technology to embed web pages.
Built in UI controls	JavaFX contains Built-in components which are not dependent on operating system. The UI component are just enough to develop a full featured application.
CSS like styling	JavaFX code can be embedded with the CSS to improve the style of the application. We can enhance the view of our application with the simple knowledge of CSS.
Swing interoperability	The JavaFX applications can be embedded with swing code using the Swing Node class. We can update the existing swing application with the powerful features of JavaFX.

Canvas API	Canvas API provides the methods for drawing directly in an area of a JavaFX scene.
Rich Set of APIs	JavaFX provides a rich set of API's to develop GUI applications.
Integrated Graphics Library	An integrated set of classes are provided to deal with 2D and 3D graphics.
Graphics Pipeline	JavaFX graphics are based on Graphics rendered pipeline(prism). It offers smooth graphics which are hardware accelerated.
High Performance Media Engine	The media pipeline supports the playback of web multimedia on a low latency. It is based on a Gstreamer Multimedia framework.
Self-contained application deployment model	Self Contained application packages have all of the application resources and a private copy of Java and JavaFX Runtime.

Architecture of JavaFX:

JavaFX has numerous built-in elements that are interconnected with each other. JavaFX library comprises a valuable collection of APIs, classes, and interfaces that are more than sufficient to produce rich internet applications and GUI applications with intense graphics that can run consistently over multiple platforms.

The subsequent figure displays the complete architecture of the JavaFX platform. Here you can examine the elements that support JavaFX API.



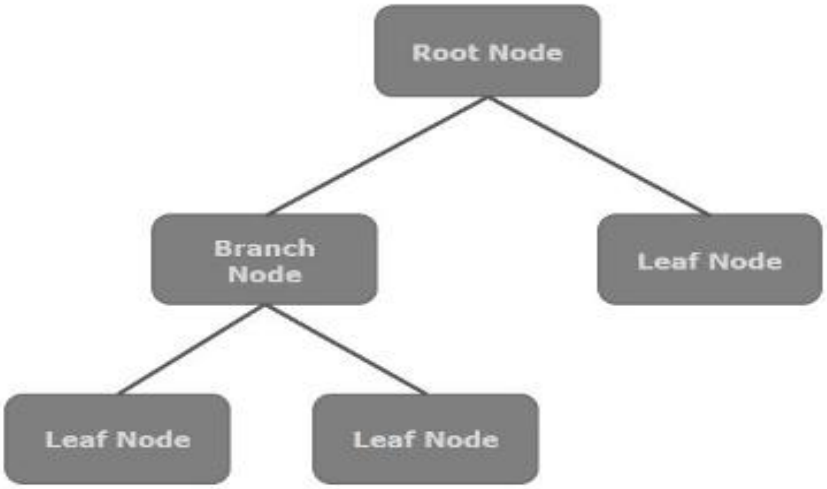
As we can see in the above figure that, JavaFX architecture comprises many different components. These components are briefly described as follows:

1. **JavaFX API –** The topmost layer of JavaFX architecture holds a JavaFX public API that implements all the required classes that are capable of producing a full-featured JavaFX application with rich graphics. The list of all the important packages of this API is as follows.
2. **javafx.animation:** It includes classes that are used to combine transition-based animations such as fill, fade, rotate, scale and translation, to the JavaFX nodes (collection of nodes makes a scene graph).
3. **javafx.css** − It comprises classes that are used to append CSS–like styling to the JavaFX GUI applications.
4. **javafx.geometry** − It contains classes that are used to represent 2D figures and execute methods on them.
5. **javafx.scene** − This package of JavaFX API implements classes and interfaces to establish the scene graph. In extension, it also renders sub-packages such as canvas, chart, control, effect, image, input, layout, media, paint, shape, text, transform, web,

etc. These are the diverse elements that sustain this precious API of JavaFX.

6. **javafx.application** − This package includes a collection of classes that are responsible for the life cycle of the JavaFX application.
7. **javafx.event** − It includes classes and interfaces that are used to perform and manage JavaFX events.
8. **javafx.stage** − This package of JavaFX API accommodates the top-level container classes used for the JavaFX application.
9. **Scene Graph –** A Scene Graph is the starting point of the development of any of the GUI Applications. In JavaFX, all the GUI Applications are made using a Scene Graph only. The Scene Graph includes the primitives of the rich internet applications that are known as nodes. The nodes are of three general types. They are as follows-

- **Root Node –** A root node is a node that does not have any node as its parent.
- **Leaf Node –** A leaf node is a node that does not contain any node as its children.
- **Branch Node –** A branch node is a node that contains a node as its parent and also has a node as its children.



10. **Quantum Toolkit –** Quantum Toolkit is used to connect prism and glass windowing tool kits collectively and makes them prepared for the above layers in the stack. In simple words, it ties Prism and GWT together and makes them available to JavaFX.

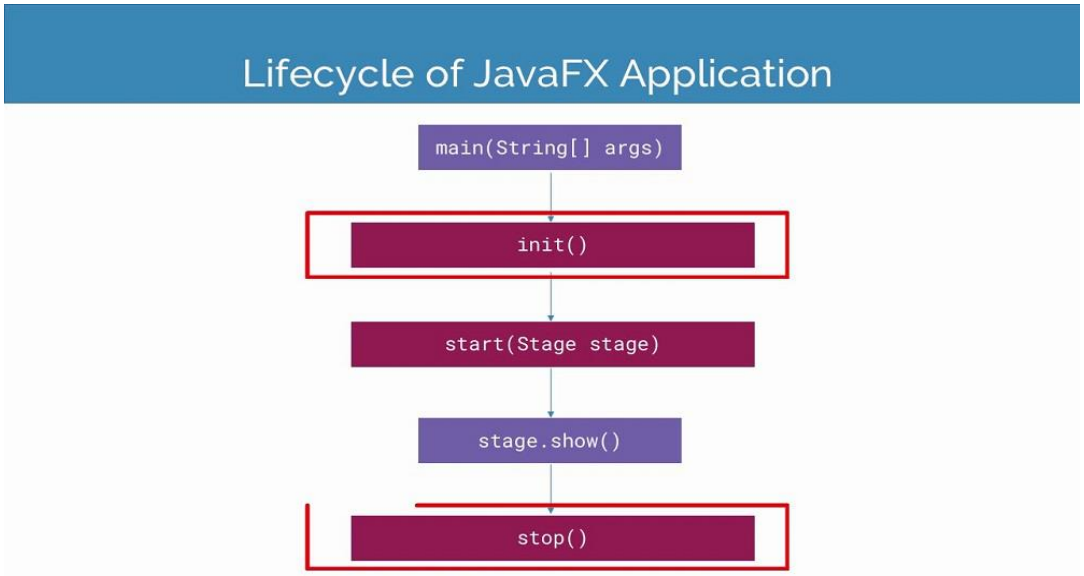
11. **Prism –** The graphics of the JavaFX applications are based on the hardware-accelerated graphics rendering pipeline, commonly known as Prism. The Prism engine supports smooth JavaFX graphics that can be executed swiftly when utilized with a backed graphics card or graphics processing unit (GPU).

12. **Glass Windowing Toolkit –** Glass Windowing Toolkit or simply Glass is a platform-dependent layer that assists in connecting the JavaFX platform to the primary operating system (OS). Glass Windowing Toolkit is very useful as it provides services such as controlling the windows, events, timers, and surfaces to the native operating system.

13. **WebView –** JavaFX applications can also insert web pages. To embed web pages, Web View of JavaFX uses a new HTML rendering engine technology known as WebKitHTML. WebView is used to make it possible to insert web pages within a JavaFX application.

14. **Media Engine –** Like the graphics pipeline, JavaFX also possesses a media pipeline that advances stable internet multimedia playback at low latency. This high-performance media engine or media pipeline is based on a multimedia framework known as Gstreamer.

LifeCycle of a JavaFX Application



There are in total three life cycle methods of a JavaFX Application class. These methods are –

- **start() –** The start() method is the entry point method of the JavaFX application where all the graphics code of JavaFX is to be written.
- **init() –** The init() method is an empty method that can be overridden. In this method, the user cannot create a stage or a scene.
- **stop() –** The stop() method is an empty method that can also be overridden, just like the init() method. In this method, the user can write the code to halt the application.

Other than these methods, the JavaFX application also implements a static method known as **launch().** This launch() method is used to launch the JavaFX application. As stated earlier, the launch() method is static, the user should call it from a static method only. Generally, that static method, which calls the launch() method, is the main() method only.

Whenever a user launches a JavaFX application, there are few actions that are carried out in a particular manner only. The following is the order given in which a JavaFX application is launched.

1. Firstly, an instance of the application class is created.
2. After that, the init() method is called.
3. After the init() method, the start() method is called.
4. After calling the start() method, the launcher waits for the JavaFX application to end and then calls the stop() method.

Terminating the JavaFX application:

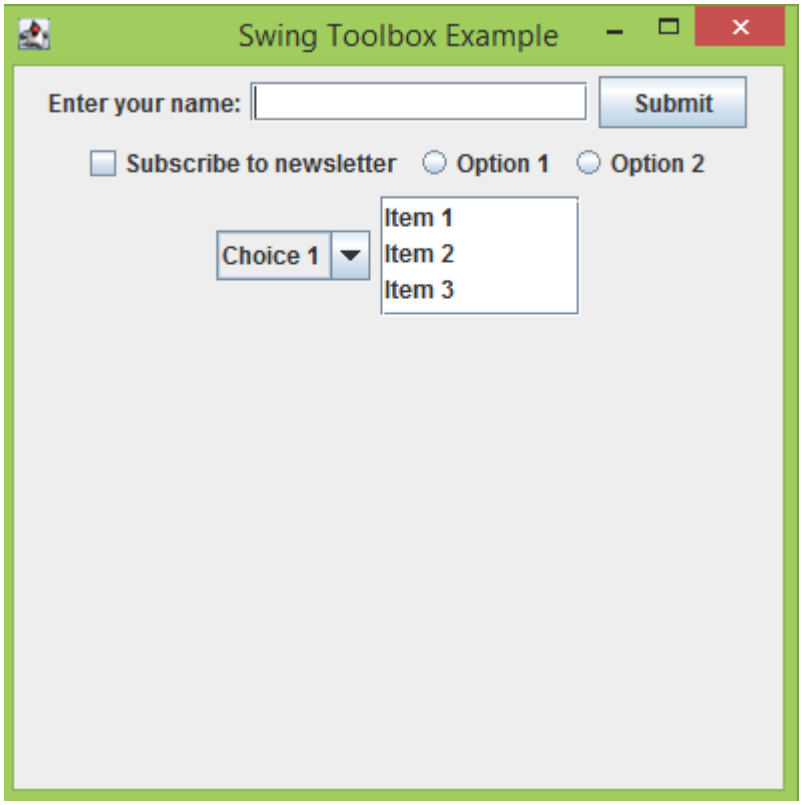
As soon as the last window of the JavaFX application is closed, the JavaFX application is stopped implicitly. The user can turn off this function by passing the Boolean value “False” to the static method **setImplicitExit().** This method should always be called from a static context only.
The user can also stop a JavaFX application explicitly by practicing any of the two methods, **Platform.exit()** or **System.exit(int)**.

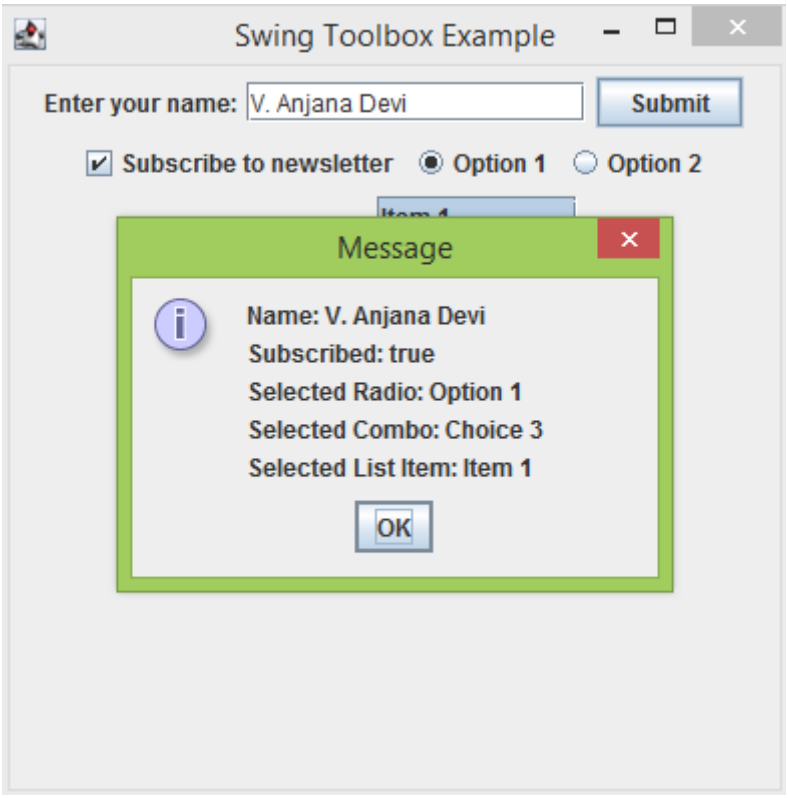
```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SwingExample {
    public static void main(String[] args) {
        // Create the main frame
        JFrame frame = new JFrame("Swing Toolbox Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 400);
```

```
        frame.setLayout(new FlowLayout());

        // Add a label
        JLabel label = new JLabel("Enter your name:");
        frame.add(label);

        // Add a text field
        JTextField textField = new JTextField(15);
        frame.add(textField);

        // Add a button
        JButton button = new JButton("Submit");
        frame.add(button);

        // Add a checkbox
        JCheckBox checkBox = new JCheckBox("Subscribe to
newsletter");
        frame.add(checkBox);

        // Add radio buttons
        JRadioButton radio1 = new JRadioButton("Option 1");
        JRadioButton radio2 = new JRadioButton("Option 2");
        ButtonGroup group = new ButtonGroup();
        group.add(radio1);
        group.add(radio2);
```

```
        frame.add(radio1);
        frame.add(radio2);

        // Add a combo box
        String[] options = {"Choice 1", "Choice 2", "Choice 3"};
        JComboBox<String> comboBox = new
JComboBox<>(options);
        frame.add(comboBox);

        // Add a list
        DefaultListModel<String> listModel = new
DefaultListModel<>();
        listModel.addElement("Item 1");
        listModel.addElement("Item 2");
        listModel.addElement("Item 3");
        JList<String> list = new JList<>(listModel);
        JScrollPane listScrollPane = new JScrollPane(list);
        listScrollPane.setPreferredSize(new Dimension(100, 60));
        frame.add(listScrollPane);

        // Add action listener to the button
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String name = textField.getText();
                boolean subscribed = checkBox.isSelected();
```

```
            String selectedOption = radio1.isSelected() ? "Option 1" :
"Option 2";
            String selectedChoice = (String)
comboBox.getSelectedItem();
            String selectedListItem = list.getSelectedValue();


            String message = "Name: " + name + "\n"
                        + "Subscribed: " + subscribed + "\n"
                        + "Selected Radio: " + selectedOption + "\n"
                        + "Selected Combo: " + selectedChoice + "\n"
                        + "Selected List Item: " + selectedListItem;


            JOptionPane.showMessageDialog(frame, message);
        }
    });

    // Show the frame
    frame.setVisible(true);
  }
}
```


Swing Toolbox Example
Enter your name:
Submit
Subscribe to newsletter
Option 1
Option 2
Choice 1
Item 1
Item 2
Item 3


Swing Toolbox Example
Enter your name:
V. Anjana Devi
Submit
Subscribe to newsletter
Option 1
Option 2
Message
Name: V. Anjana Devi
Subscribed: true
Selected Radio: Option 1
Selected Combo: Choice 3
Selected List Item: Item 1
OK