The Origins of Swing

- Swing did not exist in the early days of Java.
- The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface. One reason for the limited nature of the AWT is that it translates its various visual components into their corresponding, platform-specific equivalents, or peers. This means that the look and feel of a component is defined by the platform, not by Java. Because the AWT components use native code resources, they are referred to as heavyweight.

The use of native peers led to several problems.

- First, because of variations between operating systems, a component might look, or even act, differently on different platforms. This potential variability threatened the overarching philosophy of Java: write once, run anywhere.
- Second, the look and feel of each component was fixed (because it is defined by the platform) and could not be (easily) changed.
- Third, the use of heavyweight components caused some frustrating restrictions. For example, a heavyweight component is always rectangular and opaque.

The solution was Swing. Introduced in 1997, Swing was included as part of the Java Foundation Classes (JFC). Swing was initially available for use with Java 1.1 as a separate library. However, beginning with Java 1.2, Swing (and the rest of the JFC) was fully integrated into Java.

Swing eliminates a number of the limitations inherent in the AWT, Swing does not replace it. Instead, Swing is built on the foundation of the AWT. This is why the AWT is still a crucial part of Java. Swing also uses the same event handling mechanism as the AWT.

Difference between AWT and Swing

There are many differences between java awt and swing that are given below,

No.	Java AWT	Java Swing
1)	AWT components are **platform-dependent**.	Java swing components are **platform-independent**.
2)	AWT components are **heavyweight**.	Swing components are **lightweight**.
3)	AWT **doesn't support pluggable look and feel**.	Swing **supports pluggable look and feel**.
4)	AWT provides **less components** than Swing.	Swing provides **more powerful components** such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC**.

Two Key Swing Features

- Swing Components Are Lightweight With very few exceptions, Swing components are lightweight. This means that they are written entirely in Java and do not map directly to platform-specific peers. Because lightweight components are rendered using graphics primitives, they can be transparent, which enables nonrectangular shapes. Thus, lightweight components are more efficient and more flexible. Furthermore, because lightweight components do not translate into native peers, the look and feel of each component is determined by Swing, not by the underlying operating system.
- Swing Supports a Pluggable Look and Feel Swing supports a pluggable look and feel (PLAF). Because each Swing component is rendered by Java code rather than by native peers, the look and feel of a component is under the control of Swing. This fact means that it is possible to separate the look and feel of a component from the logic of the component, and this is what Swing does.
- **MVC Model**

The MVC Connection In general, a visual component is a composite of three distinct aspects: • The way that the component looks when rendered on the screen • The way that the component reacts to the user • The state information associated with the component.

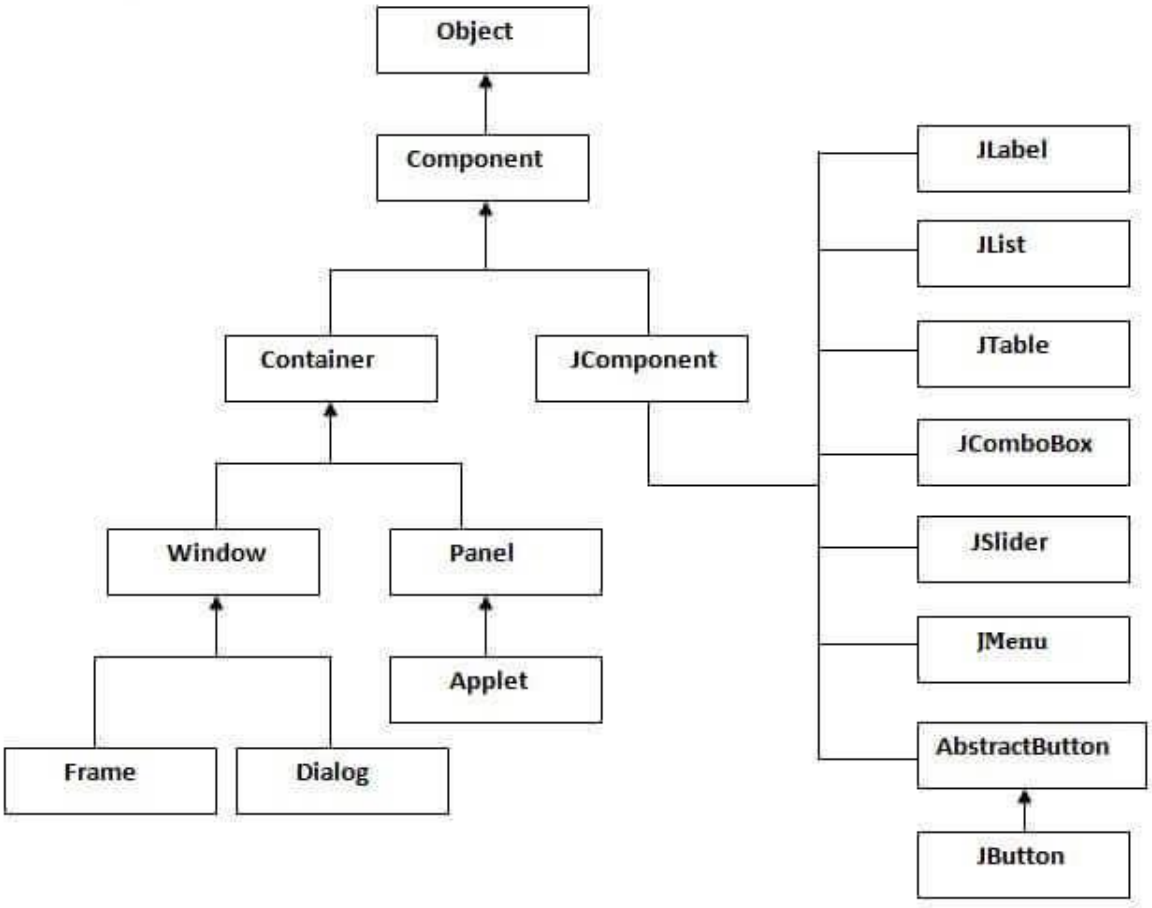
- The MVC architecture is successful because each piece of the design corresponds to an aspect of a component. In MVC terminology, the model corresponds to the state information associated with the component. For example, in the case of a check box, the model contains a field that indicates if the box is checked or unchecked. The view determines how the component is displayed on the screen, including any aspects of the view that are affected by the current state of the model. The controller determines how the component reacts to the user. For example, when the user clicks a check box, the controller reacts by changing the model to reflect the user's choice (checked or unchecked). This then results in the view being updated.

- Swing uses a modified version of MVC that combines the view and the controller into a single logical entity called the UI delegate. For this reason, Swing's approach is called either the Model-Delegate architecture or the Separable Model architecture. Therefore, although Swing's component architecture is based on MVC, it does not use a classical implementation of it. Swing's pluggable look and feel is made possible by its Model-Delegate architecture. Because the view (look) and controller (feel) are separate from the model, the look and feel can be changed without affecting how the component is used within a program. Conversely, it is possible to customize the model without affecting the way that the component appears on the screen or responds to user input. To support the Model-Delegate architecture, most Swing components contain two objects. The first represents the model. The second represents the UI delegate. Models are defined 862 Part III: Software Development Using Java by interfaces. For example, the model for a button is defined by the ButtonModel interface. UI delegates are classes that inherit ComponentUI. For example, the UI delegate for a

button is ButtonUI. Normally, your programs will not interact directly with the UI delegate.

Hierarchy of Java Swing classes

The hierarchy of java swing API is given below.



1.A Simple Swing Application Programm

```
// A simple Swing application.
import javax.swing.*;
class SwingDemo {
SwingDemo() {
// Create a new JFrame container.
JFrame jfrm = new JFrame("A Simple Swing Application");
```

```
// Give the frame an initial size.
jfrm.setSize(275, 100);
// Terminate the program when the user closes the application.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// Create a text-based label.
JLabel jlab = new JLabel(" Swing means powerful GUIs.");
// Add the label to the content pane.
jfrm.add(jlab);
// Display the frame.
jfrm.setVisible(true);
}
public static void main(String args[]) {
// Create the frame on the event dispatching thread.
SwingUtilities.invokeLater(new Runnable() {
public void run() {
new SwingDemo();
}
});
}
}
```

Explanation of Program Code:

1.It begins by creating a **JFrame**, using this line of code:

JFrame jfrm = new JFrame("A Simple Swing Application");

This creates a container called **jfrm** that defines a rectangular window complete with a title bar; close, minimize, maximize, and restore buttons; and a system menu.

2.The window is sized using this statement:

jfrm.setSize(275, 100);

The **setSize( )** method (which is inherited by **JFrame** from the AWT class **Component**) sets the dimensions of the window, which are specified in pixels. Its general form is shown here:

void setSize(int *width*, int *height*)

3.The entire application to terminate when its top-level window is closed. There are a couple of ways to achieve this. The easiest way is to call **setDefaultCloseOperation( )**, as the program does:

jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

After this call executes, closing the window causes the entire application to terminate. The general form of **setDefaultCloseOperation( )** is shown here:

void setDefaultCloseOperation(int *what*)

The value passed in ***what*** determines what happens when the window is closed.

4.The **add( )** method is inherited by **JFrame** from the AWT class **Container**.

5.SwingDemo constructor causes the window to become visible:

jfrm.setVisible(true);

The **setVisible( )** method is inherited from the AWT **Component** class. If its argument is**true**, the window will be displayed. Otherwise, it will be hidden.

6.The **SwingDemo** constructor is invoked using these lines of code:

```
SwingUtilities.invokeLater(new Runnable() {
public void run() {
new SwingDemo();
}
});
```

This sequence causes a **SwingDemo** object to be created on the ***event dispatching thread*** rather than on the main thread of the application. Here's why. In general, Swing programs are event-driven. For example, when a user interacts with a component, an event is generated. An event is passed to the application by calling an event handler defined by the application. However, the handler is executed on the event dispatching thread provided by Swing and not on the main thread of the application.

7.SwingUtilities class provide 2 methods :

static void invokeLater(Runnable *obj*)

static void invokeAndWait(Runnable *obj*)throws InterruptedException, InvocationTargetException

Here, *obj* is a **Runnable** object that will have its **run( )** method called by the event dispatching thread. The difference between the two methods is that **invokeLater( )** returns immediately, but **invokeAndWait( )** waits until **obj.run( )** returns. You can use one of these methods tocall a method that constructs the GUI for your Swing application, or whenever you need to modify the state of the GUI from code not executed by the event dispatching thread.

2.Event Handling in Swing Program

```
// Handle an event in a Swing program.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class EventDemo {
JLabel jlab;
EventDemo() {
// Create a new JFrame container.
JFrame jfrm = new JFrame("An Event Example");
// Specify FlowLayout for the layout manager.
jfrm.setLayout(new FlowLayout());
// Give the frame an initial size.
jfrm.setSize(220, 90);
// Terminate the program when the user closes the application.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// Make two buttons.
JButton jbtnAlpha = new JButton("Alpha");
JButton jbtnBeta = new JButton("Beta");
// Add action listener for Alpha.
jbtnAlpha.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent ae) {
```

```
jlab.setText("Alpha was pressed.");
}
});
// Add action listener for Beta.
jbtnBeta.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent ae) {
jlab.setText("Beta was pressed.");
}
});
// Add the buttons to the content pane.
jfrm.add(jbtnAlpha);
jfrm.add(jbtnBeta);
// Create a text-based label.
jlab = new JLabel("Press a button.");
// Add the label to the content pane.
jfrm.add(jlab);
// Display the frame.
jfrm.setVisible(true);
}
public static void main(String args[]) {
// Create the frame on the event dispatching thread.
SwingUtilities.invokeLater(new Runnable() {
public void run() {
new EventDemo();
}
});
}
}
```

3.Create a Swing Applet Program

```
// A simple Swing-based applet
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
/*
This HTML can be used to launch the applet:
<object code="MySwingApplet" width=220 height=90>
</object>
*/
public class MySwingApplet extends JApplet {
JButton jbtnAlpha;
JButton jbtnBeta;
JLabel jlab;
// Initialize the applet.
public void init() {
try {
SwingUtilities.invokeAndWait(new Runnable () {
public void run() {
makeGUI(); // initialize the GUI
}
});
} catch(Exception exc) {
System.out.println("Can't create because of "+ exc);
}
}
// This applet does not need to override start(), stop(),
// or destroy().
```

```
// Set up and initialize the GUI.
private void makeGUI() {
// Set the applet to use flow layout.
setLayout(new FlowLayout());
// Make two buttons.
jbtnAlpha = new JButton("Alpha");
jbtnBeta = new JButton("Beta");
// Add action listener for Alpha.
jbtnAlpha.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent le) {
jlab.setText("Alpha was pressed.");
}
});
// Add action listener for Beta.
jbtnBeta.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent le) {
jlab.setText("Beta was pressed.");
}
});
// Add the buttons to the content pane.
add(jbtnAlpha);
add(jbtnBeta);
// Create a text-based label.
jlab = new JLabel("Press a button.");
// Add the label to the content pane.
add(jlab);
}
}
```