RESEARCH PAPER ON SWINGS

Jaideep Verma, Inderjeet Singh Behl, Ankush Saini

Email ID- isbahl@yahoo.com
Email ID -ankushsaini90@gmail.com
Email ID- vickyjd12@gmail.com

Abstract:-

This hands-on introduction to Swing, the first in a two-part series on Swing programming, walks through the essential components in the Swing library. Java developer and Swing enthusiast Michael Abernethy guides you through the basic building blocks and then assists as you build basic but functional Swing application. Along the way you’ll learn how to use models to ease the process of dealing with the data. Most Swing developers know by now that Swing components have a separable model-and-view design. And many Swing users have run across articles saying that Swing is based on something called a "modified MVC (model-view-controller) architecture."

But accurate explanations of how Swing components are designed, and how their parts all fit together, have been hard to come by -- until now. The silence ends with the publication of this article, a major white paper on Swing component design. It provides a comprehensive technical overview of Swing's modified MVC structure and demystifies many other facets of Swing component architecture as well.

1. Introduction:-

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 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 restrictions like a heavyweight component is always rectangular and opaque. Swing was included as part of the Java Foundation Classes (JFC). Beginning with Java 1.2, Swing (and the rest of the JFC) was Swing does not replace AWT. 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.

2. History:

Swing is the primary Java GUI widget toolkit. It is part of Oracle’s Java Foundation Classes (JFC) — an API for providing a graphical user interface (GUI) for Java programs. Swing was developed to provide a more sophisticated set of GUI components than the earlier Abstract Window Toolkit (AWT). Swing provides a native look and feel that emulates the look and feel of several platforms, and also supports a pluggable look and feel that allows applications to have a look and feel unrelated to the underlying platform. It has more powerful and flexible components than AWT. In addition to familiar components such as buttons, check boxes and labels, Swing provides several advanced components such as tabbed panel, scroll panes, trees, tables, and lists. Unlike AWT components, Swing components are not implemented by platform-specific code. Instead they are written entirely in Java and therefore are platform-independent. The term "lightweight" is used to describe such an element.

3. Design Goals:

The overall goal for the Swing project was:

To build a set of extensible GUI components to enable developers to more rapidly develop powerful Java front ends for commercial applications.

To this end, the Swing team established a set of design goals early in the project that drove the resulting architecture. These guidelines mandated that Swing would:

1. Be implemented entirely in Java to promote cross-platform consistency and easier maintenance.
2. Provide a single API capable of supporting multiple look-and-feels so that developers and end-users would not be locked into a single look-and-feel.

3. Enable the power of model-driven programming without requiring it in the highest-level API.

4. Adhere to JavaBeans design principles to ensure that components behave well in IDEs and builder tools.

5. Provide compatibility with AWT APIs where there is overlapping, to leverage the AWT knowledge base and ease porting.

4. Separable model architecture:-

It is generally considered good practice to center the architecture of an application around its data rather than around its user interface. To support this paradigm, Swing defines a separate model interface for each component that has a logical data or value abstraction. This separation provides programs with the option of plugging in their own model implementations for Swing components.

The following table shows the component-to-model mapping for Swing.

<table>
<thead>
<tr>
<th>Component</th>
<th>Model Interface</th>
<th>Model Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>JButton</td>
<td>ButtonModel</td>
<td>GUI</td>
</tr>
<tr>
<td>JToggleButton</td>
<td>ButtonModel</td>
<td>GUI/data</td>
</tr>
<tr>
<td>JCheckBox</td>
<td>ButtonModel</td>
<td>GUI/data</td>
</tr>
<tr>
<td>JRadioButton</td>
<td>ButtonModel</td>
<td>GUI/data</td>
</tr>
<tr>
<td>JMenu</td>
<td>ButtonModel</td>
<td>GUI</td>
</tr>
<tr>
<td>JMenuItem</td>
<td>ButtonModel</td>
<td>GUI</td>
</tr>
<tr>
<td>JCheckBoxMenuItem</td>
<td>ButtonModel</td>
<td>GUI/data</td>
</tr>
<tr>
<td>JRadioButtonMenuItem</td>
<td>ButtonModel</td>
<td>GUI/data</td>
</tr>
<tr>
<td>JComboBox</td>
<td>ComboBoxModel</td>
<td>data</td>
</tr>
<tr>
<td>JProgressBar</td>
<td>BoundedRangeModel</td>
<td>GUI/data</td>
</tr>
<tr>
<td>JScrollBar</td>
<td>BoundedRangeModel</td>
<td>GUI/data</td>
</tr>
<tr>
<td>JSlider</td>
<td>BoundedRangeModel</td>
<td>GUI/data</td>
</tr>
<tr>
<td>JTabbedPane</td>
<td>SingleSelectionModel</td>
<td>GUI</td>
</tr>
<tr>
<td>JList</td>
<td>ListModel</td>
<td>data</td>
</tr>
<tr>
<td>JList</td>
<td>ListSelectionModel</td>
<td>GUI</td>
</tr>
<tr>
<td>JTable</td>
<td>TableModel</td>
<td>data</td>
</tr>
<tr>
<td>JTable</td>
<td>TableColumnModel</td>
<td>GUI</td>
</tr>
<tr>
<td>JTree</td>
<td>TreeModel</td>
<td>data</td>
</tr>
<tr>
<td>JTree</td>
<td>TreeSelectionModel</td>
<td>GUI</td>
</tr>
<tr>
<td>JEditorPane</td>
<td>Document</td>
<td>data</td>
</tr>
<tr>
<td>JTextPane</td>
<td>Document</td>
<td>data</td>
</tr>
<tr>
<td>JTextArea</td>
<td>Document</td>
<td>data</td>
</tr>
<tr>
<td>JPasswordField</td>
<td>Document</td>
<td>data</td>
</tr>
</tbody>
</table>
5. Lightweight notification:-

The following models in Swing use the *lightweight notification*, which is based on the `ChangeListener/ChangeEvent` API:

<table>
<thead>
<tr>
<th>Model</th>
<th>Listener</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoundedRangeModel</td>
<td>ChangeListener</td>
<td>ChangeEvent</td>
</tr>
<tr>
<td>ButtonModel</td>
<td>ChangeListener</td>
<td>ChangeEvent</td>
</tr>
<tr>
<td>SingleSelectionModel</td>
<td>ChangeListener</td>
<td>ChangeEvent</td>
</tr>
</tbody>
</table>

The `ChangeListener` interface has a single generic method:

```java
public void stateChanged(ChangeEvent e)
```

The only state in a `ChangeEvent` is the event "source." Because the source is always the same across notifications, a single instance can be used for all notifications from a particular model. Models that use this mechanism support the following methods to add and remove ChangeListeners:

```java
public void addChangeListener(ChangeListener l)
```

```java
public void removeChangeListener(ChangeListener l)
```

Therefore, to be notified when the value of a `JSlider` has changed, the code might look like this:

```java
JSlider slider = new JSlider();
BoundedRangeModel model = slider.getModel();
model.addChangeListener(new ChangeListener() {

    public void stateChanged(ChangeEvent e) {
```

```java
```
// need to query the model

// to get updated value...

BoundedRangeModel m =

(BoundedRangeModel)e.getSource();

System.out.println("model changed: "] +

m.getValue());

}
});

To provide convenience for programs that don’t wish to deal with separate model objects, some Swing component classes also provide the ability to register ChangeListeners directly on the component (so the component can listen for changes on the model internally and then propagates those events to any listeners registered directly on the component). The only difference between these notifications is that for the model case, the event source is the model instance, while for the component case, the source is the component.

So we could simplify the preceding example to:

JSlider slider = new JSlider();

slider.addChangeListener(new ChangeListener() {

public void stateChanged(ChangeEvent e) {

}});
// the source will be

// the slider this time..

JSlider s = (JSlider)e.getSource();

System.out.println("value changed: " + s.getValue());

5.1. Stateful notification:

Models that support stateful notification provide event Listener interfaces and event objects specific to their purpose. The following table shows the breakdown for those models:

<table>
<thead>
<tr>
<th>Model</th>
<th>Listener</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListModel</td>
<td>ListDataListener</td>
<td>ListDataEvent</td>
</tr>
<tr>
<td>ListSelectionModel</td>
<td>ListSelectionListener</td>
<td>ListSelectionEvent</td>
</tr>
<tr>
<td>ComboBoxModel</td>
<td>ListDataListener</td>
<td>ListDataEvent</td>
</tr>
<tr>
<td>TreeModel</td>
<td>TreeModelListener</td>
<td>TreeModelEvent</td>
</tr>
<tr>
<td>TreeSelectionModel</td>
<td>TreeSelectionListener</td>
<td>TreeSelectionEvent</td>
</tr>
<tr>
<td>TableModel</td>
<td>TableModelListener</td>
<td>TableModelEvent</td>
</tr>
<tr>
<td>TableColumnModel</td>
<td>TableColumnModelListener</td>
<td>TableColumnMod</td>
</tr>
</tbody>
</table>
The usage of these APIs is similar to the lightweight notification, except that the listener can query the event object directly to find out what has changed. For example, the following code dynamically tracks the selected item in a JList:

```java
String items[] = {"One", "Two", "Three");

JList list = new JList(items);
ListSelectionModel sModel = list.getSelectionModel();
sModel.addListSelectionListener
(normal ListSelectionListener() {

public void valueChanged(ListSelectionEvent e) {

    // get change information directly

    // from the event instance...

    if (!e.getValueIsAdjusting()) {

        System.out.println("selection changed: "+

        e.getFirstIndex());
```
6. Difference Between in AWT and Swing:-

<table>
<thead>
<tr>
<th>AWT</th>
<th>Swing</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWT stands for Abstract windows toolkit.</td>
<td>Swing is also called as JFC’s (Java Foundation classes).</td>
</tr>
<tr>
<td>AWT components are called Heavyweight component.</td>
<td>Swings are light weight component because swing components sits on the top of AWT components and do the work.</td>
</tr>
<tr>
<td>AWT components require java.awt package.</td>
<td>Swing components require javax.swing package.</td>
</tr>
<tr>
<td>AWT components are platform dependent.</td>
<td>Swing components are made in purely java and they are platform independent.</td>
</tr>
<tr>
<td>This feature is not supported in AWT.</td>
<td>We can have different look and feel in Swing.</td>
</tr>
<tr>
<td>These feature is not available in AWT.</td>
<td>Swing has many advanced features like JTable, JTabbed pane which is not available in AWT. Also, Swing components are called &quot;lightweight&quot; because they do not require a native OS object to implement their functionality. JDialog and JFrame are heavyweight, because they do have a peer. So components like JButton, JTextArea, etc., are lightweight because they do not have an OS peer.</td>
</tr>
</tbody>
</table>

With AWT, you have 21 "peers" (one for each control and one for the dialog itself). A "peer" is a widget provided by the operating system, such as a button object or an entry field object. With Swing, you would have only one peer, the operating system's window object. All of the buttons, entry fields, etc. are drawn by the Swing package on the drawing surface provided by the window object. This is the reason that Swing has more code. It has to draw the button or other control and implement its
behavior instead of relying on the host operating system to perform those functions.

<table>
<thead>
<tr>
<th>AWT is a thin layer of code on top of the OS.</th>
<th>Swing is much larger. Swing also has very much richer functionality.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using AWT, you have to implement a lot of things yourself.</td>
<td>Swing has them built in.</td>
</tr>
</tbody>
</table>

7. Relationship to AWT:-

Since early versions of Java, a portion of the Abstract Window Toolkit (AWT) has provided platform-independent APIs for user interface components. In AWT, each component is rendered and controlled by a native peer component specific to the underlying windowing system.

By contrast, Swing components are often described as lightweight because they do not require allocation of native resources in the operating system's windowing toolkit. The AWT components are referred to as heavyweight components. Much of the Swing API is generally a complementary extension of the AWT rather than a direct replacement. In fact, every Swing lightweight interface ultimately exists within an AWT heavyweight component because all of the top-level components in Swing (JApplet, JDialog, JFrame, and JWindow) extend an AWT top-level container. Prior to Java 6 Update 10, the use of both lightweight and heavyweight components within the same window was generally discouraged due to Z-order incompatibilities. However, later versions of Java have fixed these issues, and both Swing and AWT components can now be used in one GUI without Z-order issues.

The core rendering functionality used by Swing to draw its lightweight components is provided by Java 2D, another part of JFC.
8. Features of Swing:-

How to Integrate with the Desktop Class
With the Desktop class you can enable your Java application to interact with default applications associated with specific file types on the host platform.

How to Create Translucent and Shaped Windows
As of the Java Platform, Standard Edition 6 Update 10 release, you can add translucent and shaped windows to your Swing applications. This lesson shows you how.

How to Decorate Components with JLayer
JLayer is a flexible and powerful decorator for Swing components. It enables you to draw on components and respond to component events without modifying the underlying component directly.

How to Use Actions
With Action objects, you can coordinate the state and event handling of two or more components that generate action events. For example, you can use a single Action to create and coordinate a tool-bar button and a menu item that perform the same function.

How to Use Swing Timers
With the Swing Timer class, you can implement a thread that performs an action after a delay, and optionally continues to repeat the action. The action executes in the event dispatch thread.

How to Support Assistive Technologies
Swing components have built-in support for assistive technologies. Your program can provide even better support by following a few rules.

How to Use the Focus Subsystem
Some programs need to manipulate focus ???? for example, to validate input, or change the tab-order of components. This section describes some techniques you can use to customize focus in your program.

How to Use Key Bindings
With key bindings, you can specify how components react to user typing.

How to Use Modality in Dialogs
This section describes a new modality model emerged in the Java™ SE version 6 and enables you to apply different modality types to dialog boxes.
How to Print Tables
This section describes the printing capabilities for tables and explains how to add printing support to your programs.

How to Print Text
This section describes the printing capabilities for text components and explains how to add printing support to your programs.

How to Create a Splash Screen
With the SplashScreen class you can close the splash screen, change the splash-screen image, obtain the image position or size, and paint in the splash screen.

How to Use the System Tray
This section describes how to add a tray icon to the system tray and apply a text tooltip, a pop-up menu, balloon messages, and a set of listeners associated with it.

Solving Common Problems Using Other Swing Features
This section tells you how to fix problems you might encounter while trying to use the information in this lesson.

9. Advantages/Disadvantages of Swing over AWT:-

9.1. Advantages:-
Swing provides a richer set of components than AWT. They are 100% Java-based.

AWT on the other hand was developed with the mind set that if a component or capability of a component wasn't available on one platform, it won't be available on any platform. Something quickly portable from platform x, to y, to z. Due to the peer-based nature of AWT, what might work on one implementation might not work on another, as the peer-integration might not be as robust. Many of the original AWT problems were traceable to differences in peer implementations.

This is not to say that there are less bugs in Swing, though most are out these days. Its just that if a bug exists in Swing, its the same problem on all platforms, which was not the case with AWT.

Swing provides both additional components and added functionality to AWT-replacement components
Swing components can change their appearance based on the current "look and feel" library that's being used. You can use the same look and feel as the platform you're on, or use a different look and feel
Swing components follow the Model-View-Controller paradigm (MVC), and thus can provide a much more flexible UI.

Swing provides "extras" for components, such as:
- Icons on many components
- Decorative borders for components
- Tooltips for components
Swing components are lightweight (less resource intensive than AWT)
Swing provides built-in double buffering
Swing provides paint debugging support for when you build your own components

9.2. Disadvantages:

It requires Java 2 or a separate JAR file
If you're not very careful when programming, it can be slower than AWT (all components are drawn)
Swing components that look like native components might not act exactly like native components

10. Using Swing Components: Examples

The table that follows lists every example in the Using Swing Components lesson, with links to required files and to where each example is discussed. The first column of the table has links to JNLP files that let you run the examples using Java™ Web Start.

NOTE: Release 7.0 is required to run all applets and Java Web Start examples. Most examples will run on an earlier release but you must compile and run them locally.

To run an example using Java Web Start, click the [Launch] link in the first column of the table. The first time you run an example, there will be a delay while Java Web Start downloads the JAR file containing the class files for this lesson's examples. Afterward, the examples should execute more quickly.

10.2. Compiling and Running the Examples Locally

The second column in the table below has links to zip files for each demo that you can open and run in the NetBeans IDE.

If you download an individual example, take care to have all the necessary files in the proper hierarchy when you compile and run it. All of the examples in the Swing tutorial are placed in a package. For example, the components examples are placed in a components package. See the following image for the complete structure. Note that any examples using images expect their image files to be in a directory named images that is in the same directory as the example's src files.

Here is a typical setup:
You can find out which files each example needs by consulting the following table or by looking at the comments at the beginning of each source file.

11. References:-


[3]. books.google.co.in/


[5]. http://www.becbapatla.ac.in/cse/


[7]. http://www.withoutbook.com/