UNIT V – COLLECTIONS FRAMEWORK

Collection overview – Recent changes to collection - Collection interface – Collection classes – Working with maps –Collection algorithms - The legacy classes and interfaces. Applet class: Types – Basics – Architecture – Skeleton – Display methods – repainting – Status window – HTML applet tag – Passing parameter - Creating a swing applet - Painting in swing - A paint example, Exploring swing

---

Creating a swing applet

The second type of program that commonly uses Swing is the applet. Swing-based applets are similar to AWT-based applets, but with an important difference: A Swing applet extends JApplet rather than Applet. JApplet is derived from Applet. Thus, JApplet includes all of the functionality found in Applet and adds support for Swing. JApplet is a top-level Swing container, which means that it is not derived from JComponent. Because JApplet is a top-level container, it includes the various panes described earlier. This means that all components are added to JApplet's content pane in the same way that components are added to JFrame's content pane.

Swing applets use the same four lifecycle methods as described in Chapter 21: init( ), start( ), stop( ), and destroy( ). Of course, you need override only those methods that are needed by your applet. Painting is accomplished differently in Swing than it is in the AWT, and a Swing applet will not normally override the paint( ) method.

One other point: All interaction with components in a Swing applet must take place on the event dispatching thread, as described in the previous section. This threading issue applies to all Swing programs.

Example of a Swing applet.

It provides the same functionality as the previous application, but does so in applet form.

```
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;
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);
        }
}
private void makeGUI()
{
        setLayout(new FlowLayout());
        jbtnAlpha = new JButton("Alpha");
        jbtnBeta = new JButton("Beta");
        jbtnAlpha.addActionListener(new ActionListener()
        {
                public void actionPerformed(ActionEvent le)
                {
                        jlab.setText("Alpha was pressed.");
                }
        });
        jbtnBeta.addActionListener(new ActionListener()
        {
                public void actionPerformed(ActionEvent le)
                {
                        jlab.setText("Beta was pressed.");
                }
        });
```

```
            add(jbtnAlpha);
            add(jbtnBeta);
            jlab = new JLabel("Press a button.");
            add(jlab);
        }
}
```

Output:

There are two important things to notice about this applet. First, MySwingApplet extends JApplet. As explained, all Swing-based applets extend JApplet rather than Applet. Second, the init( ) method initializes the Swing components on the event dispatching thread by setting up a call to makeGUI( ). Notice that this is accomplished through the use of invokeAndWait( ) rather than invokeLater( ). Applets must use invokeAndWait( ) because the init( ) method must not return until the entire initialization process has been completed. In essence, the start( ) method cannot be called until after initialization, which means that the GUI must be fully constructed.

Inside makeGUI( ), the two buttons and label are created, and the action listeners are added to the buttons. Finally, the components are added to the content pane. Although this example is quite simple, this same general approach must be used when building any Swing GUI that will be used by an applet.

Painting in Swing

Although the Swing component set is quite powerful, you are not limited to using it because Swing also lets you write directly into the display area of a frame, panel, or one of Swing's other components, such as JLabel. Although many (perhaps most) uses of Swing will not involve drawing directly to the surface of a component, it is available for those applications that need this capability. To write output directly to the surface of a component, you will use one or more drawing methods defined by the AWT, such as drawLine( ) or drawRect( ).

Painting Fundamentals

Swing's approach to painting is built on the original AWT-based mechanism, but Swing's implementation offers more finally grained control. Before examining the specifics of Swing-based painting, it is useful to review the AWT-based mechanism that underlies it.

The AWT class Component defines a method called paint( ) that is used to draw output directly to the surface of a component. For the most part, paint( ) is not called by your program.

Rather, paint( ) is called by the run-time system whenever a component must be rendered. This situation can occur for several reasons. For example, the window in which the component is displayed can be overwritten by another window and then uncovered. Or, the window might be minimized and then restored. The paint( ) method is also called when a program begins running. When writing AWT-based code, an application will override paint( ) when it needs to write output directly to the surface of the component.

Because JComponent inherits Component, all Swing's lightweight components inherit the paint( ) method. However, you will not override it to paint directly to the surface of a component. The reason is that Swing uses a bit more sophisticated approach to painting that involves three distinct methods: paintComponent( ), paintBorder( ), and paintChildren( ). These methods paint the indicated portion of a component and divide the painting process into its three distinct, logical actions. In a lightweight component, the original AWT method paint( ) simply executes calls to these methods, in the order just shown.

To paint to the surface of a Swing component, you will create a subclass of the component and then override its paintComponent( ) method. This is the method that paints the interior of the component. You will not normally override the other two painting methods. When overriding paintComponent( ), the first thing you must do is call super.paintComponent( ), so that the superclass portion of the painting process takes place. (The only time this is not required is when you are taking complete, manual control over how a component is displayed.) After that, write the output that you want to display. The paintComponent( ) method is shown here:

protected void paintComponent(Graphics g)

The parameter g is the graphics context to which output is written. To cause a component to be painted under program control, call repaint( ). It works in Swing just as it does for the AWT. The repaint( ) method is defined by Component. Calling it causes the system to call paint( ) as soon as it is possible to do so. Because painting is a time-consuming operation, this mechanism allows the run-time system to defer painting momentarily until some higher-priority task has completed, for example. Of course, inSwing the call to paint( ) results in a call to paintComponent( ). Therefore, to output to the surface of a component, your program will store the output until paintComponent( ) is called. Inside the override paintComponent( ), you will draw the stored output.

Compute the Paintable Area

When drawing to the surface of a component, you must be careful to restrict your output to the area that is inside the border. Although Swing automatically clips any output that will exceed the boundaries of a component, it is still possible to paint into the border, which will then get overwritten when the border is drawn. To avoid this, you must compute the paintable area of the component. This is the area defined by the current size of the component minus the space used by the border. Therefore, before you paint to a component, you must obtain the width of the border and then adjust your drawing accordingly. To obtain the border width, call getInsets( ), shown here:

Insets getInsets( )

This method is defined by Container and overridden by JComponent. It returns an Insets object that contains the dimensions of the border. The inset values can be obtained by using these fields:

```
int top;
int bottom;
int left;
int right;
```

These values are then used to compute the drawing area given the width and the height of the component. You can obtain the width and height of the component by calling getWidth( )
and getHeight( ) on the component. They are shown here:

```
int getWidth( )
int getHeight( )
```

By subtracting the value of the insets, you can compute the usable width and height of the Component.
