

---

## INSTALLING AND USING THE JFC

---

All of the Swing classes are in 3 jar files, called `swing.jar`, `swingall.jar` and `windows.jar`. If you are using Java 1.1, you can download the Swing classes from [java.sun.com](http://java.sun.com) site and install them by simply unzipping the downloaded file. It is important that your `CLASSPATH` variable contain the paths for these three jar files.

```
set CLASSPATH=.;d:\java11\lib\classes.zip;d:\swing\swing.jar;
d:\swing\windows.jar;d:\swing\swingall.jar;
```

All programs which are to make use of the JFC, must import the following files:

```
//swing classes
import com.sun.java.swing.*;
import com.sun.java.swing.event.*;
import com.sun.java.swing.border.*;
import com.sun.java.swing.text.*;
```

In the Java JDK 1.2, these change to “`javax.swing`”

```
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;
import javax.swing.text.*;
```

and so forth.

### ***Ideas Behind Swing***

The Swing components are referred to as “lightweight” components, because they don’t rely on native user-interface components. They are, in fact, 100% pure Java. Thus, a Swing `JButton` does not rely on a Windows button or a Motif button or a Mac button to implement its functionality. They also use fewer classes to achieve this interface than the previous heavier-weight awt classes. In addition, there are many more Swing user-interface components than there were awt components. Swing gives us image buttons, hover buttons, tooltips, tables, trees, splitter panels, customizable dialog boxes and quite a few other components.

Since Swing components create their look and feel completely within the Swing class hierarchy, you can have a pluggable look and feel to emulate Windows, Motif, Macintosh or the native Swing look.

In addition, Swing components make use of an architecture derived from the model-view-controller design pattern we discussed in the first chapter. The idea of this MVC pattern you recall, is to keep the data in a *model* class, display the data in a *view* class and vary the data and view using a *controller* class. We'll see that this is exactly how the JList and Jtable handle their data.

### ***The Swing Class Hierarchy***

All Swing components inherit from the JComponent class. While JComponent is much like Component in its position in the hierarchy, JComponent is the level that provides the pluggable look and feel. It also provides

- Keystroke handling that works with nested components.
- A border property that defines both the border and the component's insets.
- Tooltips that pop up when the mouse hovers over the component.
- Automatic scrolling of any component when placed in a scroller container.

Because of this interaction with the user interface environment, Swing's JComponent is actually more like the awt's Canvas than its Component class.

---

## WRITING A SIMPLE JFC PROGRAM

---

Getting started using the Swing classes is pretty simple. Application windows inherit from `JFrame` and applets inherit from `JApplet`. The only difference between `Frame` and `JFrame` is that you cannot add components or set the layout directly for `JFrame`. Instead, you must use the `getContentPane` method to obtain the container where you can add components and vary the layout.

```
getContentPane().setLayout(new BorderLayout());
JButton b = new JButton("Hi");
GetContentPane().add(b);           //add button to layout
```

This is sometimes a bit tedious to type each time, so we recommend creating a simple `JPanel` and adding it to the `JFrame` and then adding all the components to that panel.

```
JPanel jp = new JPanel();
getContentPane().add(jp);
JButton b = new JButton("Hi");
jp.add(b);
```

`JPanels` are containers much like the `awt Panel` object, except that they are automatically double buffered and repaint more quickly and smoothly.

### ***Setting the Look and Feel***

If you do nothing, Swing programs will start up in their own native look and feel rather than the Windows, Motif or Mac look. You must specifically set the look and feel in each program, using a simple method like the following:

```
private void setLF()
{
    // Force to come up in the System L&F
    String laf = UIManager.getSystemLookAndFeelClassName();
    try {
        UIManager.setLookAndFeel(laf);
    }
    catch (UnsupportedLookAndFeelException exc)
        {System.err.println("Unsupported: " + laf);}
    catch (Exception exc)
        {System.err.println("Error loading " + laf);}
}
```

The system that generates one of several look and feels and returns self-consistent classes of visual objects for a given look and feel is an example of an Abstract Factory pattern as we discussed in the previous chapter.

### **Setting the Window Close Box**

Like the Frame component, the system exit procedure is *not* called automatically when a user clicks on the close box. In order to enable that behavior, you must add a WindowListener to the frame and catch the WindowClosing event. This can be done most effectively by subclassing the WindowAdapter class:

```
private void setCloseClick()
{
    //create window listener to respond to window close click
    addWindowListener(new WindowAdapter()
    {
        public void windowClosing(WindowEvent e)
        {System.exit(0);}
    });
}
```

### **Making a JxFrame Class**

Since we must always set the look and feel and must always create a WindowAdapter to close the JFrame, we have created a JxFrame class which contains those two functions, and which calls them as part of initialization:

```
public class JxFrame extends JFrame
{
    public JxFrame(String title)
    {
        super(title);
        setCloseClick();
        setLF();
    }
}
```

The *setLF* and *setCloseClick* methods are included as well. It is this JxFrame class that we use in virtually all of our examples in this book, to avoid continually retyping the same code.

### **A Simple Two Button Program**

Now with these fundamentals taken care of, we can write a simple program with two buttons in a frame.



One button switches the color of the background and the other causes the program to exit. We start by initializing our GUI and catching both button clicks in an *actionPerformed* method:

```
public class SimpleJFC extends JFrame
    implements ActionListener
{
    JButton OK, Quit;          //these are the buttons
    JPanel jp;                //main panel
    Color color;              //background color
    //-----
    public SimpleJFC()        {
        super("Simple JFC Program");
        color = Color.yellow; //start in yellow
        setGUI();
    }
    //-----
    private void setGUI()    {
        jp = new JPanel();    //central panel
        getContentPane().add(jp);

        //create and add buttons
        OK = new JButton("OK");
        Quit = new JButton("Quit");
        OK.addActionListener(this);
        Quit.addActionListener(this);
        jp.add(OK);
        jp.add(Quit);

        setSize(new Dimension(250,100));
        setVisible(true);
    }
    //-----
    public void actionPerformed(ActionEvent e)    {
        Object obj = e.getSource();
        if(obj == OK)
            switchColors();
        if(obj == Quit)
            System.exit(0);
    }
}
```

The only remaining part is the code that switches the background colors. This is, of course, extremely simple as well:

```

private void switchColors()    {
    if(color == Color.green)
        color = Color.yellow;
    else
        color = Color.green;
    jp.setBackground(color);
    repaint();
}

```

That's all there is to writing a basic JFC application. JFC applets are identical except for the applet's *init* routine replacing the constructor. Now let's look at some of the power of the more common JFC components.

### More on JButtons

The JButton has several constructors to specify text, an icon or both:

```

JButton(String text);
JButton(Icon icon);
JButton(String text, Icon icon);

```

You can also set two other images to go with the button

```

setSelectedIcon(Icon icon);    //shown when clicked
setRolloverIcon(Icon icon);    //shown when mouse over

```

Finally, like all other JComponents, you can use *setToolTipText* to set the text of a Tooltip to be displayed when the mouse hovers over the button. The code for implementing these small improvements is simply

```

OK = new JButton("OK",new ImageIcon("color.gif"));
OK.setRolloverIcon(new ImageIcon("overColor.gif"));
OK.setToolTipText("Change background color");
Quit = new JButton("Quit", new ImageIcon("exit.gif"));
Quit.setToolTipText("Exit from program");

```

The resulting application window is shown below.

