Still More Swing

Icons

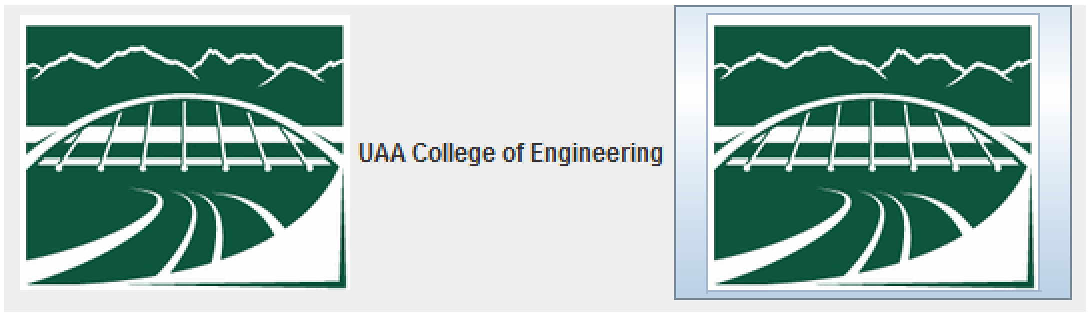
An icon is simply a small picture, although it doesn't have to be small. Labels, buttons, or menu items may have just a string displayed on it, just an icon, both, or nothing at all. An icon is an instance of the ImageIcon class and is based on a digital image file in a standard format such as .png, .gif, .jpg, etc.

In the example below, the file "COENG.gif" is in the default directory for the Java program.

We can load it into an ImageIcon, then assign the ImageIcon to a Java component that allows us to set the icon:

```
import javax.swing.ImageIcon;
…
        setLayout(new FlowLayout());

        ImageIcon coeng = new ImageIcon("COENG.gif");
        JLabel uaaLabel = new JLabel("UAA College of Engineering");
        uaaLabel.setIcon(coeng);
        this.add(uaaLabel);

        JButton btn = new JButton(coeng);
        this.add(btn);
```

Output:

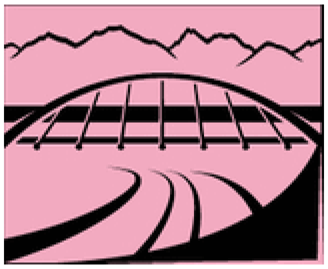


Some developers like to use JLabel with ImageIcon to display pictures/images on the JFrame with no text. If you want to add text later you can use the setText() method.

The same technique works for a JMenuItem. If you have a button or menu item with only an icon and no text, then this is where you would use setActionCommand so your action listener can distinguish between what component initiated the action.

Changing Visibility

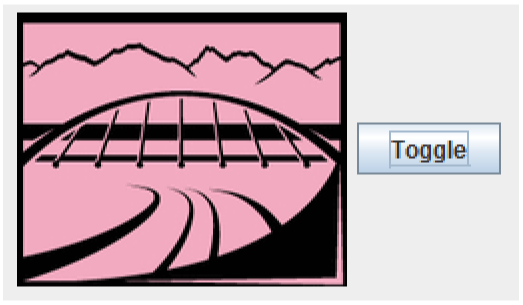
Sometimes it is useful to change the visibility of a component – i.e. make it invisible or visible again. We can do this by calling the setVisible method. If we have an image with COENG as in the previous example, and another named COENG-Invert as shown below:



The following code will create two JLabels, one with the icon set for each image. However, we use a flag to make one invisible and toggle between the two. If the sizes are identical and the components are next to each other in the layout then there will be a seamless transition between the two images.

```
private JLabel coeng1;
private JLabel coeng2;
private boolean showing1 = true;

public GUI_App()
{
    super();
    setSize(810,640);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLayout(new FlowLayout());

    ImageIcon icon1 = new ImageIcon("COENG.gif");
    ImageIcon icon2 = new ImageIcon("COENG-Invert.gif");

    coeng1 = new JLabel(icon1);
    coeng2 = new JLabel(icon2);

    coeng1.setVisible(true);
    coeng2.setVisible(false);
    showing1 = true;

    this.add(coeng1);
    this.add(coeng2);

    JButton btn = new JButton("Toggle");
    btn.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e) {
            if (showing1)
            {
                coeng1.setVisible(false);
                coeng2.setVisible(true);
                showing1=false;
            }
```

```
                    else
                    {
                        coeng1.setVisible(true);
                        coeng2.setVisible(false);
                        showing1=true;
                    }
                }
            });
            this.add(btn);
            setVisible(true);
        }
```



We could achieve a similar effect with a single JLabel but changing the icon inside with the setIcon method.

paint() vs paintComponent()

So far we have been overriding the paint() method to do our custom painting. When we do this for a JFrame, it is the only way to go, as this is the only method available to override to perform our custom rendering.

However, if we desire a custom paint job for a Component (e.g. a JPanel) then generally we want to override paintComponent instead of paint. The mechanism looks identical except we use paintComponent instead of paint:

```
public void paintComponent(Graphics g)
{
    super.paintComponent(g);
    g.fillRect(0,0,200,200);
}
```

So what is the reason for this?

Swing further factors the paint() call into three separate methods, which are invoked in the following order:

```
protected void paintComponent(Graphics g)
protected void paintBorder(Graphics g)
protected void paintChildren(Graphics g)
```

There is generally no reason to mess with rendering the border or children. So there are efficiencies to be gained by overriding only the paintComponent method.

There is also a big difference in terms of order. Any custom code you write in a paint() method is run after the parent paint() is completed – meaning your paint code runs after paintComponent/Border/Children so it will be rendered after (and on top of) any rendering for the children.

If you override paintComponent then your code is run before paintChildren is called, so children will be rendered after (and on top of) your code.

See this code for an example:

```
public class GUI_App extends JFrame
{
    private class MyPanel extends JPanel
    {
       public MyPanel()
       {
           setLayout(new GridLayout(2,1));
           //setLayout(new FlowLayout()); Experiment changing
           JPanel pan1 = new JPanel();
           pan1.setBackground(Color.black);
           JPanel pan2 = new JPanel();
           pan2.setBackground(Color.red);
           this.add(pan1);
           this.add(pan2);
       }

       /*                               Experiment changing
       public void paint(Graphics g)
       {
           super.paint(g);
           g.fillRect(0,0,200,200);
       }
       */
       /*                               Experiment changing */
       public void paintComponent(Graphics g)
       {
           super.paintComponent(g);
           g.fillRect(0,0,200,200);
       }

    }

    public GUI_App()
    {
        super();
        setSize(810,640);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new BorderLayout());
```

```
            MyPanel p = new MyPanel();
            this.add(p,BorderLayout.CENTER);

            setVisible(true);
        }
        public static void main(String[] args) {
            GUI_App app = new GUI_App();
        }
    }
```

If you swap between the paint and paintComponent methods you will see that paint will go on top of the colored JPanel child components, but paintComponent will be underneath the child JPanels. You can see it "underneath" if you switch to a FlowLayout or a null layout where you specify the dimensions of the JPanels so part of the rectangle can be seen underneath.

Scroll Bars

We saw previously that you can add any component to a JScrollPane which you can in turn add to your layout and it will display scroll bars around your component. The default is sufficient for many applications but sometimes you want to tweak the scroll settings.

```
            setLayout(new FlowLayout());

            JTextArea text = new JTextArea("Hello",5,10);
            JScrollPane scrollPane = new JScrollPane(text);
            this.add(scrollPane);
```

The most typical is setting the scroll bar policies. The following specify whether the horizontal or vertical scroll bars will always be present or if they are visible only when needed:

```
    scrollPane.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBA
    R_ALWAYS);

    scrollPane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_AL
    WAYS);
```

We can programmatically get and set the scrollbar positions. The following example uses a button to advance the Y scroll position.

```
            JTextArea text = new JTextArea(5,30);
            String s = "";
            for (int i = 0; i < 30; i++)
            {
                s += i + " this is a line of text" + "\n";
            }
            text.setText(s);
            JScrollPane scrollPane = new JScrollPane(text);
            this.add(scrollPane);
```

```
        JButton btn = new JButton("Scroll Text");
        btn.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent e) {
                 JScrollBar vert = scrollPane.getVerticalScrollBar();
                 if (num < vert.getMaximum())
                 {
                     num+=1;
                     vert.setValue(num);
                 }
            }
        });
        this.add(btn);
```

Miscellaneous Windowing Components

Here are some miscellaneous Swing components you may find useful:

JFileChooser – this pops up a dialog for you to choose a file. This example filters only .gif and jpg files:

```
    JFileChooser chooser = new JFileChooser();
    FileNameExtensionFilter filter = new FileNameExtensionFilter(
        "JPG & GIF Images", "jpg", "gif");
    chooser.setFileFilter(filter);
    int returnVal = chooser.showOpenDialog(parent); //parent=window obj
    if(returnVal == JFileChooser.APPROVE_OPTION)
    {
       System.out.println("You chose to open this file: " +
            chooser.getSelectedFile().getName());
    }
```

JColorChooser – see https://docs.oracle.com/javase/tutorial/uiswing/examples/components/ColorChooserDemoProject/src/components/ColorChooserDemo.java for an example, but we create a JColorChooser and implement ChangeListener which invokes stateChanged when a color is chosen

JOptionPane – showInputDialog can be used to pop up a dialog to read a string.

    String name = JOptionPane.showInputDialog(frame, "What's your name?");

JOptionPane – showMessageDialog can be used to pop up a dialog box with a string

    JOptionPane.showMessageDialog(null, “String here”);

    Null can be replaced by your frame

The JOptionPane dialogs are useful for debugging but you should refrain from heavy use in an application as they can be rather annoying and prone to error in input. A better technique is to design input into your main window instead.

More Layouts

Here are a few more layouts that you may find useful. See https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html for a complete list. We won't cover the GridBagLayout or GroupLayout or SpringLayout (the last two are mainly for GUI builders).

Box Layout

The Box layout lets us align components either vertically or horizontally. To use it invoke the constructor:

BoxLayout(Container, axis)

Container is a component like a JPanel. If we want to get the container for a JFrame, use the method getContentPane().

The axis parameter is either BoxLayout.X_AXIS, .Y_AXIS, .LINE_AXIS, or .PAGE_AXIS.

```
public GUI_App()
{
    super();
    setSize(810,640);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLayout(new
         BoxLayout(this.getContentPane(),BoxLayout.Y_AXIS));

    this.add(new JButton("Button 1"));
    this.add(new JButton("Button 2"));
    this.add(new JButton("Button 3"));
    setVisible(true);
}
```

Card Layout

The Card layout arranges components like a deck of cards. Only the "top" component is visible at a time. Usually the components are JPanels. Normally there is some controlling component like a menu that selects the top component. Another option is to use a tabbed pane, which we will cover later.

```
public GUI_App()
{
    super();
    setSize(810,640);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLayout(new BorderLayout());
    JPanel mainPane = new JPanel();
    mainPane.setLayout(new CardLayout());

    // Create the "Cards"
    JPanel card1 = new JPanel();
    card1.setBackground(Color.red);
    mainPane.add(card1, "Panel 1");
    JPanel card2 = new JPanel();
    card2.setBackground(Color.blue);
    mainPane.add(card2, "Panel 2");
    JPanel card3 = new JPanel();
    card3.setBackground(Color.green);
    mainPane.add(card3, "Panel 3");
    this.add(mainPane, BorderLayout.CENTER);

    // Controlling button
    JButton btn = new JButton("Next");
    btn.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e)
        {
            CardLayout layout = (CardLayout)
                     mainPane.getLayout();
            layout.next(mainPane);
            //layout.show(mainPane, "Panel 3");
            // Goes to panel 3
        }
    });
    this.add(btn, BorderLayout.SOUTH);

    setVisible(true);
}
```