

Swing from A to Z

Transparency and Preferred Size

By [Richard G. Baldwin](#)

Java Programming, Lecture Notes # 1015

August 14, 2000

- [Preface](#)
- [Introduction](#)
- [Sample Program](#)
- [Interesting Code Fragments](#)
- [Summary](#)
- [Complete Program Listing](#)

Preface

This series of lessons entitled *Swing from A to Z*, discusses the capabilities and features of Swing in quite a lot of detail. This series is intended for those persons who need to really understand what Swing is all about.

Recommended supplementary reading

It is recommended that in addition to studying this set of lessons, you also study my earlier lessons on Swing. A list of some of my Swing lessons can be found in an earlier [lesson](#) in this series. The lessons themselves can be found at *Baldwin's Java Programming [Tutorials](#)*.

The earlier lessons will introduce you to the use of Swing while avoiding much of the detail included in this series.

Introduction

Properties, events, and methods

In an earlier lesson, I provided lists of *properties*, *events*, and *methods* that are defined in **JComponent** and its superclasses: **Container**, **Component**, and **Object**.

Default appearance and behavior

I explained that because most Swing components extend **JComponent** either directly or indirectly, the properties, events, and methods defined in these classes provide the default appearance and behavior of most of the Swing components.

Understanding common properties, events, and methods

I also explained that the next few lessons would concentrate on an understanding of these common properties, events, and methods in order to provide an overall knowledge of the appearance and behavior of Swing components.

Will discuss specialized appearance and behavior later

After I have illustrated this common appearance and behavior of Swing components, I will embark on a study of the additional specialized appearance and behavior associated with individual components.

What's in this lesson?

This lesson emphasizes an understanding of the *opaque* and *preferredSize* properties that are common to most Swing components.

I will also illustrate several other properties, including the following:

- layout
- background
- foreground
- title
- size
- visible
- text
- `contentPane`
- source

*(Note that not all of these properties are defined in **JComponent** and its superclasses. Some are specialized properties of specific Swing components.)*

All but *contentPane* should be familiar

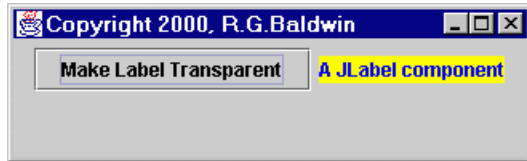
Except for *contentPane*, none of these properties are peculiar to Swing. They are also commonly used with the AWT.

Sample Program

The name of the sample program is **Swing12**. As mentioned above, this program is designed specifically to illustrate the use and behavior of the *opaque* and *preferredSize* properties.

A screen shot

A screen shot of the GUI that is produced when the program is started is shown below.



When the program is started, a **JFrame** object, about 330 pixels wide, appears on the screen. It contains a **JButton** component and a **JLabel** component in a container being managed by a **FlowLayout** manager.

FlowLayout manager

One of the characteristics of the **FlowLayout** manager is that it attempts to honor the *preferredSize* of each component in both the horizontal and vertical dimensions.

Other layout managers don't necessarily honor the *preferredSize* in either or both dimensions. Some honor one dimension, others honor the other dimension, and some (such as **GridLayout**) don't honor either dimension.

I discuss the behavior of several layout managers in some of my other [Tutorials](#).

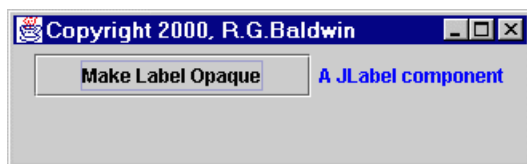
Initial states of the label and button

Initially at startup, the label contains blue text on an opaque yellow background as shown in the above screen shot.

The text showing initially on the face of the button reads "Make Label Transparent."

Click the button

When the button is clicked, the appearance of the GUI changes to that shown below.



Basically, two things happen when the button is clicked.

1. The background of the label becomes transparent, allowing the gray background of the **JFrame** object to show through. This is accomplished by setting the *opaque* property of the label to false. As a result, the yellow background disappears.

2. The text on the face of the button changes to a shorter text string ("Make Label Opaque"). However, the size of the button doesn't change. The *preferredSize* property of the button is used to cause the size of the button to remain constant regardless of the length of its text string.

Click the button again

If the button is clicked again, the appearance of the GUI reverts back to its original state.

Successively clicking the button causes the GUI to toggle between these two states.

Interesting Code Fragments

I will discuss the program in fragments. A complete listing of the program is provided in Figure 9 near the end of the lesson.

The controlling class

Figure 1 shows the beginning of the controlling class, along with the declaration and population of two reference variables, one for the button and one for the label.

```
class Swing12 extends JFrame{
    JButton button = new JButton(
        "Make Label Transparent");
    JLabel label = new JLabel(
        "A JLabel component");
```

Figure 1

Extends JFrame

Note that the controlling class extends **JFrame**. Therefore, an object of the controlling class is a GUI that can be displayed directly on the desktop.

The main() method

Figure 2 shows the **main()** method for the application. This method simply instantiates an object of the controlling class, causing a **JFrame** object to appear on the screen.

```
public static void main(String args[]) {
    new Swing12();
} //end main()
```

Figure 2

The constructor

Figure 3 shows the beginning of the constructor. This fragment sets the values of three properties.

```
Swing12(){//constructor

    getContentPane().setLayout(
        new FlowLayout());
    label.setBackground(Color.yellow);
    label.setForeground(Color.blue);
```

Figure 3

The layout property

First, the code in Figure 3 sets the *layout* property of the *content pane* of the **JFrame** object to **FlowLayout**.

This layout manager places components in the container from left to right, top to bottom, while attempting to honor the *preferredSize* value of each component.

The default preferred size

Each component has a default preferred size, and it differs from one type of component to another.

The default preferred size for both a **JButton** and a **JLabel** is a size that reasonably accommodates the text showing on the button or on the label.

The *preferredSize* property is easy to control in Swing

Unlike the AWT, it is possible to modify the *preferredSize* property of a Swing component without a requirement to extend the component.

Overriding `getPreferredSize()` in the AWT

With the AWT, the only way that I have found to control the preferred size of a component is to extend the component and override the `getPreferredSize()` method.

By doing that, it is possible to return a value for preferred size that is different from the default value.

Swing has a *preferredSize* property

Swing has a *setter* method for the preferred size property that makes it easy to change the value of the property.

Changing value may not result in a change in size

Be aware, however, that many layout managers will ignore the value of the *preferredSize* property when placing the component in a container.

Transparency is controlled by the *opaque* property

The transparency, or lack thereof, of the background of a Swing component is controlled by the value of the *opaque* property.

If this property has a value of true, the background is opaque (not transparent).

If the value of the property is false, the background is transparent, allowing whatever is behind the component to show through.

Default values for *opaque* property

The default value of the *opaque* property of a Swing **JLabel** is false.

By default, the background of the label is transparent allowing whatever is behind the **JLabel** object to show through.

Text in the label is opaque

Only the text characters are opaque by default.

If the *opaque* property is set to true, the default background color of the **JLabel** appears to be gray, so it still looks to be transparent against a gray container background.

Default transparency for a **JButton**

On the other hand, the default value of the *opaque* property of a Swing **JButton** is true.

When instantiated, the default color of the **JButton** is gray.

JButton can become transparent

If the value of the *opaque* property of a **JButton** is set to false, the background becomes transparent allowing whatever is behind the button to show through. Only the text and the border of the button remain opaque.

Make it happen

Figure 4 shows the statement that overrides the default and causes the background of the label to be opaque when the program starts running.

```
label.setOpaque(true);
```

Figure 4

Typical JavaBeans *setter* method

This is a typical *setter* method for a JavaBean property.

The name of the property

Note that the name of the property in this case is *opaque* with a lower-case "o". (I explained the property naming conventions of JavaBean properties in an earlier lesson.)

Some ordinary code

The code in Figure 5 is rather ordinary,

- Adding the button and the label to the *contentPane* of the **JFrame** object, and
- Setting values for the properties named *title*, *size*, and *visible*.

```
getContentPane().add(button);
getContentPane().add(label);
setTitle("Copyright 2000, R.G.Baldwin");
setSize(329,100);
setVisible(true);
```

Figure 5

What is `getContentPane()`?

I explained the requirement to use `getContentPane()` in an earlier lesson on Swing. You must invoke this method whenever you need to add a component to a **JFrame**, or whenever you need to set the layout manager for a **JFrame**.

The `preferredSize` setter method

As I mentioned earlier, unlike the AWT, Swing makes it possible to control the value of the *preferredSize* property using an ordinary *setter* method, as shown in Figure 6.

```
button.setPreferredSize(
    button.getSize());
```

Figure 6

I planned to change the *text* property value

In this case, I planned to change the value of the *text* property for the button at runtime. (The *text* property controls the string value displayed on the face of the **JButton**).

Didn't want the size of the button to change

I wanted to make certain that the size of the button did not change when I changed the value of the *text* property.

I knew that the initial value of the *text* property (the text showing on the button at startup) required more physical space for display than would be required for the display of later values of the *text* property.

Used startup button size as preferred size

Therefore, I used the *getter* method shown in Figure 6 to get the current value of the *size* property of the button after the GUI first became visible.

I used that value to set the *preferredSize* property for the button, causing the actual size of the button to remain constant from that point forward, even though the value of the *text* property changed to a value requiring less physical space for display.

Order of these operations is important

The order of operations was important here. It was necessary to set the *visible* property of the **JFrame** object to true before getting the value of the *size* property of the button. Until that time, the *getter* method for the *size* property of the button returned zero for both width and height. Apparently the size property is not set until the GUI is actually rendered on the screen.

An anonymous inner class action listener

This program uses an anonymous inner class, containing an **if/else** statement, as an action listener registered on the button. This action listener toggles the GUI between its two states.

The *if* clause

The first half of that action listener, including the **if** clause, is shown in Figure 7.

```
button.addActionListener(
    new ActionListener(){
        public void actionPerformed(
            ActionEvent e){
            if(label.isOpaque()){
                label.setOpaque(false);
                label.repaint();//render it
                ((JButton)e.getSource()).
                    setText("Make Label Opaque");
            }//end if
        }
    }
);
```

Figure 7

The **else** clause is shown later in Figure 8.

Use a getter method of type *is...*

Now please refer back to the **if** clause of Figure 7. The code in the **actionPerformed()** method of Figure 7 uses **label.isOpaque()** as the conditional expression in the **if** statement. This is an optional form of a *getter* method that can be used for boolean properties.

Toggle the value of the *opaque* property

If the conditional expression returns true, the value of the *opaque* property of the label is set to **false**, causing the background of the label to become transparent (not opaque).

Request a repaint for the label

The **repaint()** method is used to send a message to the operating system asking it to render the new transparent representation of the area of the screen occupied by the label.

Get a reference to the button

Then the **getSource()** method of the incoming **ActionEvent** object is used to get a reference to the **JButton** component that fired the event.

Modify the value of the *text* property

This reference is used to modify the value of the *text* property of the button.

Downcast is necessary

Note that it is necessary to downcast the reference to the source of the event before accessing the *text* property. The **getSource()** method returns a reference to the source of the event as type **Object**.

Text on button is automatically repainted

It is interesting to note that the text on the face of the button is automatically repainted without the necessity of a call to **repaint()**. However, it is necessary to call **repaint()** to cause the background of the label to be repainted.

I'm sure that there is a rule for this somewhere, but I haven't found it yet.

The else clause

Figure 8 shows the **else** clause of the **if/else** statement. This clause is executed if the value of the *opaque* property is false (meaning that the background of the label is transparent).

```
else{//if it is not opaque
    label.setOpaque(true);
    label.repaint();//render it
```

```
((JButton)e.getSource()).
    setText("Make Label Transparent");
} //end else
} //end actionPerformed
} //end ActionListener
} //end addActionListener
```

Figure 8

Reverse the previous action

The code in Figure 8 reverses the action of the **if** clause in Figure 7, returning the values of the *opaque* property of the label and the *text* property of the button to their original values.

Toggle each time the button is clicked

The effect is to cause the GUI to toggle back and forth between two states each time the button is clicked.

The remaining code

The remaining code, which you can view in Figure 9, is uninteresting. It consists simply of an anonymous inner class used to terminate the program when the user closes the **JFrame** object.

Summary

The purpose

The primary purpose of this lesson was to illustrate the use of the *opaque* and *preferredSize* properties of Swing components.

Using the *opaque* property

I have illustrated how to use the *opaque* property to control the background transparency of a **JLabel** object.

The approach used here should work with any Swing component that supports transparency.

Using the *preferredSize* property

I have also illustrated how to use the *preferredSize* property to control the size of a **JButton** object.

This approach should also work with any Swing component that supports the *preferredSize* property.

Layout manager may not cooperate

Note, however, that even though a Swing component may support the *preferredSize* property, the *layout* property of the container that contains that component may specify a layout manager that won't honor the preferred size even if the component supports it.

For example, the **GridLayout** manager doesn't support the preferred size in either dimension. Rather, all components in a **GridLayout** are rendered the same size regardless of their preferred size.

Several other properties were illustrated

In the process of illustrating the use of the *opaque* and *preferredSize* properties, I have illustrated several other properties as well.

Complete Program Listing

Figure 9 contains a complete listing of the program discussed in this lesson.

```
/*File Swing12
Rev 3/28/00
Copyright 2000, R.G.Baldwin

Illustrates the manipulation of several
properties of a JLabel and a JButton under
program control. Primary properties
manipulated are:

opaque (transparency)
preferredSize

Other properties manipulated are:

layout
background
foreground
title
size
visible
text
contentPane
source

Tested using JDK 1.2.2 under WinNT 4.0 WkStn
*****/

import java.awt.*;
import java.awt.event.*;
import javax.swing.*; //jdk 1.2 version

class Swing12 extends JFrame{
    JButton button = new JButton(
        "Make Label Transparent");
    JLabel label = new JLabel(
        "A JLabel component");
    //-----//

    public static void main(String args[]) {
        new Swing12();
    }
}
```

```

} //end main()
//-----//

Swing12(){ //constructor

    getContentPane().setLayout(
        new FlowLayout());
    label.setBackground(Color.yellow);
    label.setForeground(Color.blue);
    //Default is transparent
    label.setOpaque(true);

    getContentPane().add(button);
    getContentPane().add(label);
    setTitle("Copyright 2000, R.G.Baldwin");
    setSize(329,100);
    setVisible(true);

    //Cause the button to stay the same
    // size when its text changes
    button.setPreferredSize(
        button.getSize());

    //.....//
    //Anonymous action listener class
    button.addActionListener(
        new ActionListener(){
            public void actionPerformed(
                ActionEvent e){
                if(label.isOpaque()){
                    label.setOpaque(false);
                    label.repaint();//render it
                    ((JButton)e.getSource()).
                        setText("Make Label Opaque");
                } //end if
                else{//if it is not opaque
                    label.setOpaque(true);
                    label.repaint();//render it
                    ((JButton)e.getSource()).
                        setText(
                            "Make Label Transparent");
                } //end else
            } //end actionPerformed
        } //end ActionListener
    ); //end addActionListener

    //.....//
    //Anonymous inner terminator class
    this.addWindowListener(
        new WindowAdapter(){
            public void windowClosing(
                WindowEvent e){
                System.exit(0);
            } //end windowClosing()
        } //end WindowAdapter
    ); //end addWindowListener
} //end constructor

//-----//
} //end class Swing12

```

Figure 9

Copyright 2000, Richard G. Baldwin. Reproduction in whole or in part in any form or medium without express written permission from Richard Baldwin is prohibited.

About the author

Richard Baldwin is a college professor and private consultant whose primary focus is a combination of Java and XML. In addition to the many platform-independent benefits of Java applications, he believes that a combination of Java and XML will become the primary driving force in the delivery of structured information on the Web.

Richard has participated in numerous consulting projects involving Java, XML, or a combination of the two. He frequently provides onsite Java and/or XML training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Java Programming Tutorials, which has gained a worldwide following among experienced and aspiring Java programmers. He has also published articles on Java Programming in Java Pro magazine.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

baldwin.richard@iname.com

-end-