Swing from A to Z

The border Property

Part 6, The BorderFactory Class

By Richard G. Baldwin

Java Programming, Lecture Notes # 1025

September 25, 2000

Preface

This lesson is Part 6 in a miniseries of several parts designed to illustrate the ***border*** property and the use of that property to construct fancy borders on Swing components.

It is strongly recommended that you study the previous parts beginning with The border Property, Part 1, EtchedBorder before embarking on this lesson.

I also recommended that in addition to studying this set of lessons, you also study my earlier lessons on Swing, which are available at Gamelan. A consolidated index to those earlier lessons is available at my personal web site.

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different figures and listings while you are reading about them.

Introduction

The BorderFactory class

This lesson illustrates the use of class methods of the **BorderFactory** class to create nested compound borders.

Highlight and shadows

The lesson also illustrates the ability to specify shadow and highlight colors for 3D borders, and shows how the human eye can be tricked by specifying those colors inappropriately.

Conditioning

In particular, it is suggested that (for persons conditioned to the normal Windows scheme of displaying 3D components), simply reversing the highlight and shadow colors can cause a LOWERED component to look like a RAISED component.

Disclaimer

However, since the display of 3D components on a 2D screen is an optical illusion anyway, this reversal might not have the same effect on all viewers.

Sample Program

The name of the sample program that I will discuss to illustrate the use of the **BorderFactory** class is **Swing19**.

A screen shot

A screen shot of the GUI that is produced when the program is started is shown in Figure 1..

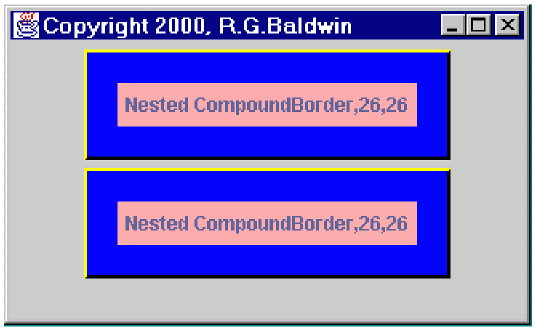
Copyright 2000, R.G.Baldwin
Nested CompoundBorder,26,26
Nested CompoundBorder,26,26

Figure 1A screen shot of Swing19.

Comparison with earlier program

For comparison purposes, the screen shot in Figure 2 shows a similar GUI that was produced in an earlier lesson by the program named **Swing14**.

The important thing to pay attention to, and to compare with the screen shot in Figure 1, is the color of the highlight and shadow on the edges of the components.

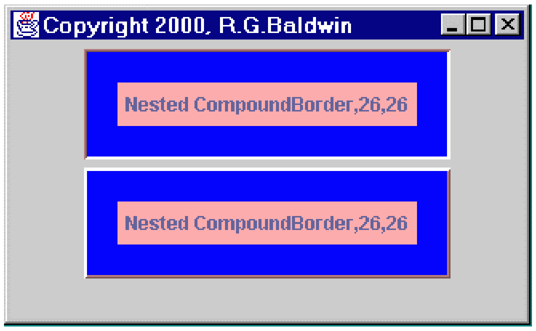
Copyright 2000, R.G.Baldwin
Nested CompoundBorder,26,26
Nested CompoundBorder,26,26

Figure 2 A screen shot of Swing14.

Two JLabel objects

Both programs create and display two different **JLabel** objects, applying a different border style to each of them.

Differences between the two programs

The primary differences between the two screen shots are:

- In **Swing14**, the *default* colors for highlight and shadow were applied to both components. In **Swing19**, I specified the highlight and shadow colors as yellow and black.
- For the top component in **Swing19**, I (inappropriately) used a bright color (yellow) for the shadow and a dark color (black) for the highlight. This caused the component to appear to protrude out of the screen instead of being depressed into the screen, even though the LOWERED version of a **BevelBorder** was used.
- The borders on the components in **Swing19** were produced using the **BorderFactory** class instead of the **new** operator, but this shouldn't produce any noticeable difference between the two screen shots.

A side trip into optical illusions

One day several years ago, I was printing some screen shots of Java GUIs produced on my Windows 95 system. I was watching the paper emerge from the printer, and from my vantage point, I was viewing the paper upside down.

Buttons were depressed

I noticed that all of the buttons appeared to be depressed into the paper rather than protruding from the paper.

Suddenly, it dawned on me that I had become conditioned to a specific stimulus just like the famous scientist's dog that salivated each time the bell rang. (I have forgotten the name of the famous scientist but I still remember the dog. So much for fame.)

Conditioned by Microsoft Windows

I had become conditioned to respond appropriately to Windows 3D component representations with highlights and shadows based on a light source coming from the upper left corner of the screen.

What does the light source do?

A light source at this location would cause the top and the left side of a protruding object to be brighter than the face of the object. It would also cause the bottom and right side of the object to be less bright than the face of the object.

An optical illusion

This is an optical illusion that causes a 2D representation to look like a 3D object.

When viewed right side up (after proper conditioning), if the left and top are brighter, the illusion is that the component is protruding from the surface. However, when viewed upside down, the illusion is that the component is depressed into the surface.

It usually isn't easy to view the screen upside down, but it is easy to view a hard-copy screen shot upside down.

Another way to destroy the illusion

Another way to destroy the illusion is to switch the brightness of the colors used for highlight and shadow, as illustrated in the previous screen shot of the top component for the program named **Swing19**.

Interesting Code Fragments

I will discuss the program named **Swing19** in fragments. A complete listing of the program is shown in Listing 4 near the end of the lesson.

Will skip material discussed earlier

Swing19 is very similar to the program named **Swing14** that I discussed in detail in an earlier lesson. Therefore, I will skip those parts of the program that were discussed in the previous lesson.

The top Swing component

Listing 1 shows the code fragment that prepares the border for the top Swing component in the screen shot for **Swing19**.

```
  CompoundBorder theBorder =
  BorderFactory.createCompoundBorder(
   BorderFactory.createBevelBorder(
   BevelBorder.LOWERED,
                Color.black,Color.yellow),
       BorderFactory.createCompoundBorder(
        BorderFactory.createMatteBorder(
         19,19,19,19,Color.blue),
          BorderFactory.createEmptyBorder(
                               5,5,5,5)));
```

Listing 1

This statement creates an object of the **CompoundBorder** class, which will be used later as the border for a **JLabel** object. (The portion of the statement that is highlighted in red will be discussed later.)

A CompoundBorder object

The programs in earlier lessons created borders using statements incorporating the **new** operator, such as the following:

new CompoundBorder(new BevelBorder(...

Swing19 uses factory methods

However, **Swing19** doesn't use the **new** operator to instantiate **Border** objects. Instead, it invokes factory methods with names like **createCompoundBorder()** to instantiate and return references to **Border** objects. These factory methods are class methods of the **BorderFactory** class.

What does Sun have to say?

Here is what Sun has to say about the **BorderFactory** class.

Factory class for vending standard Border objects. Wherever possible, this factory will hand out references to shared Border instances.

What does Flanagan have to say?

Here is what David Flanagan has to say about the **BorderFactory** class in his excellent book, Java Foundation Classes in a Nutshell.

The static methods of this class return various types of **Border** objects. These methods may return previously created shared objects, making their use more memory-efficient than creating unshared **Border** objects with the **new** operator.

The bottom line on BorderFactory

So, the bottom line seems to be that you can create your **Border** objects in either of two ways

- Using the **new** operator with the constructor for a specific **Border** class.
- Using the factory method of the **BorderFactory** class for the type of border that you need.

The use of the factory methods of the **BorderFactory** class may produce more memory-efficient programs.

Highlights and shadows

One of the constructors for the **BevelBorder** class is as follows. Here, we are interested mainly in the parameters.

```
public BevelBorder(int bevelType,
          Color highlight,
          Color shadow)
```

Creates a bevel border with the specified type, highlight and shadow colors.
  Parameters:
    bevelType - the type of bevel for the border
    highlight - the **color** to use for the bevel **highlight**
    shadow - the **color** to use for the bevel **shadow**

(This constructor is mirrored in the factory method for an object of the **BevelBorder** class.)

Parameters specify highlight and shadow colors

As you can see, the second parameter is the highlight color and the third parameter is the shadow color. Normally, we would expect the highlight to be brighter than the shadow.

What happens if you switch them?

A portion of the previous code fragment is repeated in Listing 2. (This is the portion that is highlighted in red in Listing 1.)

```
  BorderFactory.createBevelBorder(
    BevelBorder.LOWERED,
          Color.black,Color.yellow),


Listing 2
```

As you can see, this fragment requests a LOWERED **BevelBorder** object with a black highlight and a yellow shadow (the shadow was purposely made brighter than the highlight).

Optical illusion is reversed

The result is to reverse the optical illusion, causing the component to appear to be RAISED instead of LOWERED (see the top component in Figure 1, which shows a screen shot for **Swing19**).

(As mentioned earlier, since this is an optical illusion anyway, it may not appear the same to all observers.)

The bottom component

Listing 3 shows the code fragment that prepares the border for the bottom component in the screen shot for **Swing19** (see Figure 1).

```
    theBorder =
     BorderFactory.createCompoundBorder(
      BorderFactory.createBevelBorder(
             BevelBorder.RAISED,
                 Color.yellow,Color.black),
        BorderFactory.createCompoundBorder(
         BorderFactory.createMatteBorder(
          19,19,19,19,Color.blue),
           BorderFactory.createEmptyBorder(
                                 5,5,5,5)));


Listing 3
```

In this case, I caused the highlight (yellow) to be brighter than the shadow (black) so that the resulting component appeared to be RAISED as specified.

Top and bottom components look alike

Going back to the screen shot, the top component with reversed highlight and shadow looks just like the bottom component with proper highlight and shadow. The moral to the story is, be careful when you specify highlight and shadow colors.

Summary

The factory methods of the **BorderFactory** class can be used to produce **Border** objects that are shared, and therefore can be more memory-efficient than their counterparts instantiated using the **new** operator.

You can specify the highlight and shadow colors for various **Border** objects. However, you need to be careful when you do. Otherwise, you may spoil the 3D optical illusion for many observers.

Where To From Here?

That completes our miniseries on the **border** property. In the next lesson, I will begin a discussion of the **alignment** property.

Complete Program Listing

A complete listing of the program is shown in Listing 4.

```
/*File Swing19
Rev 3/30/00
Copyright 2000, R.G.Baldwin

Illustrates the use of the BorderFactory
class for nesting of CompoundBorder
objects.  This program creates and
displays two different border styles.

Borders created using BorderFactory are
shared among objects.  That is the advantage
of using the factory method.

Also illustrates specifying the highlight
and shadow colors for a BevelBorder and,
just for fun, shows that simply reversing
the two (making the shadow light and the
highlight dark) reverses the optical
illusion and causes a LOWERED 3-D component
to appear to be RAISED.  This illustrates
how we have become conditioned to having
the light source at the upper left.  The
reversal of the two colors would be correct
for a LOWERED 3-D component if the light
source were at the bottom right.

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


import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;


class Swing19 extends JFrame{

  //---------------------------------------//

  public static void main(String args[]) {
      new Swing19();
  }//end main()
  //---------------------------------------//

  //The purpose of this method is to create
  // and return an opaque pink JLabel with
  // a border.  The text content of the
  // lable is provided as the first
  // parameter.  The border type is provided
  // as the second parameter.  When the
  // label is displayed, the left and top
  // insets are displayed following the
  // text content of the label.
  JLabel makeLabel(
           String content,Border borderType){

    JLabel label = new JLabel();
    label.setBorder(borderType);
    label.setOpaque(true);
    label.setBackground(Color.pink);

    label.setText(content + ","
    +label.getInsets().left + ","
    +label.getInsets().top);

    return label;

  }//end makeLabel()
  //---------------------------------------//

  Swing19(){//constructor

    getContentPane().setLayout(
                           new FlowLayout());

    CompoundBorder theBorder =
     BorderFactory.createCompoundBorder(
      BorderFactory.createBevelBorder(
       BevelBorder.LOWERED,Color.black,
                               Color.yellow),
        BorderFactory.createCompoundBorder(
         BorderFactory.createMatteBorder(
          19,19,19,19,Color.blue),
           BorderFactory.createEmptyBorder(
                                  5,5,5,5)));

    getContentPane().add(makeLabel(
         "Nested CompoundBorder",theBorder));

    theBorder =
     BorderFactory.createCompoundBorder(
      BorderFactory.createBevelBorder(
       BevelBorder.RAISED,Color.yellow,
                               Color.black),
        BorderFactory.createCompoundBorder(
         BorderFactory.createMatteBorder(
          19,19,19,19,Color.blue),
           BorderFactory.createEmptyBorder(
                                  5,5,5,5)));

    getContentPane().add(makeLabel(
         "Nested CompoundBorder",theBorder));

    setTitle("Copyright 2000, R.G.Baldwin");
    setSize(329,200);
    setVisible(true);

    //...................................//
    //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 Swing19
```

Listing 4

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-