Swing from A to Z

The border Property

Part 3, CompoundBorder

By Richard G. Baldwin

Java Programming, Lecture Notes # 1022

September 4, 2000

Preface

This lesson is Part 3 of several parts designed to illustrate the *border* property and the use of that property to construct fancy borders on Swing components.

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, without losing your place, while you are reading about them.

Recommended supplementary reading

It is strongly recommended that you study The border Property, Part 1, Etched Border and The border Property, Part 2, BevelBorder and EmptyBorder before embarking on this lesson.

Additional supplementary material on Swing is available at *Baldwin's Java Programming Tutorials*.

Introduction

What's in this lesson?

This is the third of several sequential lessons that emphasize an understanding of the *border* property along with the use of that property to construct components having different border styles.

What was in the previous parts?

Part 1 set the background for future discussions, and dealt specifically with the use of the **EtchedBorder** class. Part 2 continued the discussion adding three more border styles.

So far, I have discussed the following border styles.

- EtchedBorder
- BevelBorder RAISED
- BevelBorder LOWERED
- EmptyBorder

A cosmetic problem

I pointed out that when the first three border styles listed above were applied to a **JLabel** object, the results weren't very attractive because the ends of the text were too close to the edge of the **JLabel** object.

I also pointed out that the solution to the problem described above involves the combined use of the **EmptyBorder** and the **CompoundBorder** classes.

What is a CompoundBorder?

Here is part of what Sun has to say about the **CompoundBorder** class.

> A composite Border class used to compose two Border objects into a single border by nesting an inside Border object within the insets of an outside Border object.
>
> For example, this class may be used to add blank margin space to a component with an existing decorative border:

Can be nested

Compound borders themselves can also be nested.

That is to say, a compound border object can be used as one of the two border objects specified for a new compound border object. That object can be included in another compound border object, etc.

What are the long-term plans?

This lesson will introduce the **CompoundBorder**, and will illustrate four different border styles using **CompoundBorder** that are similar to, but much more attractive than those illustrated in the earlier lessons.

Subsequent lessons will illustrate four additional styles that make use of **EmptyBorder** and **CompoundBorder** to enhance their attractiveness.

In addition, one of the lessons in this miniseries on borders will illustrate the use of nested **CompoundBorder** objects.

Sample Program

A screen shot

The name of the sample program that I will discuss to illustrate borders is **Swing13**.

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

If you have sufficient screen space, you may find it useful to open a second copy of this lesson in another browser window so that you can view this screen shot while reading the remaining material.

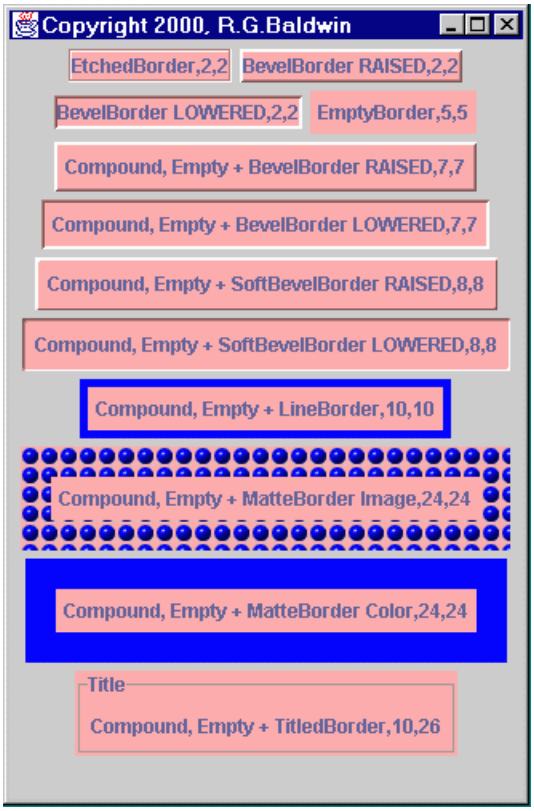

Figure 1 A Screen Shot of the Running Program

Twelve JLabel objects

The program creates and displays twelve different **JLabel** objects, applying a different border style to each of them.

See rows 3, 4, 5, and 6 of the screen shot

In this lesson, I will be concentrating on the components appearing on the third, fourth, fifth, and sixth rows.

Combine EmptyBorder with another border

Each of these components uses a **CompoundBorder** object to combine an **EmptyBorder** object with another border object.

The use of the **EmptyBorder** object provides a margin of blank space between the text and the visible border, greatly improving the appearance of the component.

Interesting Code Fragments

I will discuss the program in fragments. A complete listing of the program is provided in Listing 3 near the end of this lesson. I will skip those parts of the program that were discussed in the previous lessons

Compound Empty + BevelBorder RAISED,7,7

Listing 1 shows the code fragment that caused the label in the third row of the screen shot to be added to the *contentPane* of the **JFrame** container.

```
    getContentPane().add(makeLabel(
    "Compound, Empty + BevelBorder RAISED",
    new CompoundBorder(
       new BevelBorder(
            BevelBorder.RAISED),
          new EmptyBorder(5,5,5,5))));

Listing 1
```

The specified text content

The first parameter to the **makeLabel()** method is highlighted in Italics in the code fragment of Listing 1. This parameter eventually becomes part of the text content of the label.

The specified border style

The second parameter to **makeLabel()** (highlighted in boldface in Listing 1) is a reference to a new object of the class **CompoundBorder**.

Construction of the CompoundBorder object

The actual construction of a compound border is achieved by passing appropriate parameters to the constructor for the **CompoundBorder** object.

The outside border is a raised BevelBorder

The first parameter (shown in red) to the constructor for **CompoundBorder** is a reference to a **BevelBorder** object that serves as the outside border. In this case, I caused it to be RAISED.

This is the same border that I illustrated in an earlier lesson, which appears on the right side of the first row of the screen shot.

A margin of blank space

The second parameter to the constructor for **CompoundBorder** (shown in blue) is a reference to a new **EmptyBorder** object that serves as the inside border. In this case, the **EmptyBorder** object produces a five-pixel margin of blank space around the text.

A more attractive label

In my opinion, this is a much more attractive component than the one in the right side of the first row (of the screen shot) that doesn't have the blank space. Of course, this is a matter of opinion.

The inset values

The inset of the left border and the inset of the top border each have a value of seven pixels for this **CompoundBorder**. This is the sum of the five-pixel margin that I specified and the two pixels normally required for the **BevelBorder**.

Code for three more labels

Listing 2 shows the code that was used to produce the components in the fourth, fifth, and sixth rows of the screen shot shown earlier.

```
    getContentPane().add(makeLabel(
     "Compound, Empty + BevelBorder LOWERED",
       new CompoundBorder(
         new BevelBorder(
           BevelBorder.LOWERED),
             new EmptyBorder(5,5,5,5))));


    getContentPane().add(makeLabel(
      "Compound, Empty + SoftBevelBorder " +
      "RAISED",
     new CompoundBorder(
         new SoftBevelBorder(
           SoftBevelBorder.RAISED),
             new EmptyBorder(5,5,5,5))));


    getContentPane().add(makeLabel(
      "Compound, Empty + SoftBevelBorder " +
      "LOWERED",
        new CompoundBorder(
          new SoftBevelBorder(
            SoftBevelBorder.LOWERED),
              new EmptyBorder(5,5,5,5))));

Listing 2
```

The compound lowered BevelBorder

The component in the fourth row is just like the one discussed above (in the third row), except that it is LOWERED instead of RAISED.

As discussed above, this border also has inset values of seven pixels on the left and top border sections.

SoftBevelBorder

Another of the standard border styles that is available is the **SoftBevelBorder**. Here is what Sun has to say about this border style.

A class which implements a raised or lowered bevel with softened corners.

Compound SoftBevelBorder components

The components in the fifth and sixth rows are similar to the previous two except that they use a **SoftBevelBorder** instead of a **BevelBorder**.

Interestingly, the **SoftBevelBorder** has an inset value of three pixels instead of two pixels as is the case with **BevelBorder**. This results in total inset values of eight pixels for the compound border.

How do they differ?

I couldn't visually discern much difference between the regular and soft bevel styles on my high-resolution monitor. Being curious, I used an image editor to enlarge the screen shot for the left end of the components in the third, fourth, fifth, and sixth rows just to see how the "softening" was accomplished.

In case you are interested in such things, that enlarged image is shown in Figure 2.

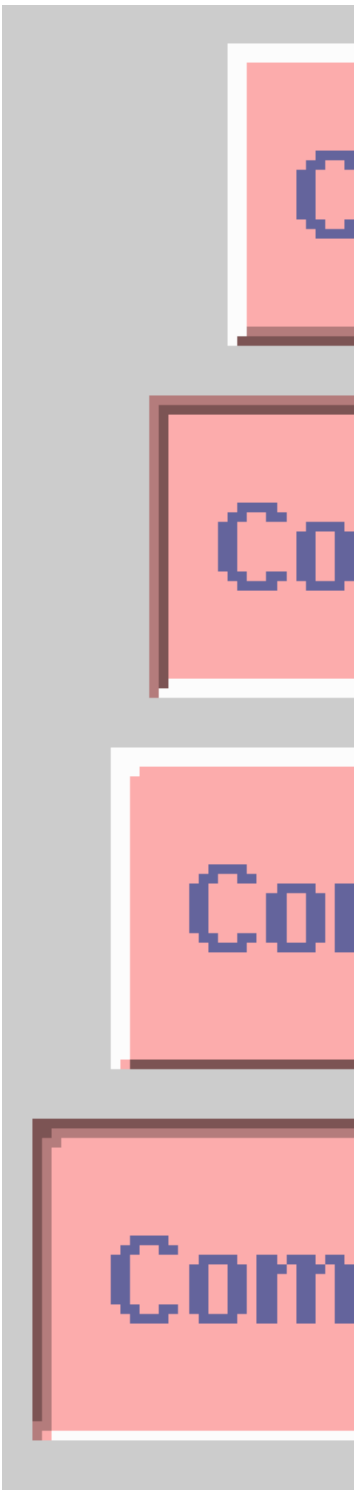

Figure 2 An Enlarged View

I find this sort of thing interesting because the entire process of rendering three-dimensional pictures on a two-dimensional screen is an optical illusion anyway. I often like to look at enlarged versions to see just how the optical illusion is created.

Summary

In this lesson, I introduced you to the **CompoundBorder** class as a way to combine two border styles to produce a new border style. One of the two combined borders serves as an inside border, and the other serves as an outside border.

Any two **Border** objects can be combined in this way, and either of them can be a **CompoundBorder** object, leading to a nesting capability.

In this lesson, I caused the inside border to be an **EmptyBorder** object for the purpose of providing a band of blank space surrounding the text of the **JLabel** component.

Where To From Here?

I have three more border styles to illustrate in this miniseries on borders:

- LineBorder
- MatteBorder
- TitleBorder

I will illustrate two different variations on **MatteBorder**. One creates a border by using icons as tiles. The other forms the border using a solid, non transparent color.

After illustrating and discussing these three, I will illustrate the nesting of **CompoundBorder** objects.

Complete Program Listing

A complete listing of the program is shown in Listing 3.

```
/*File Swing13
Rev 3/28/00
Copyright 2000, R.G.Baldwin


Illustrates the border property.  This
program creates and displays several
different border types surrounding a
JLabel object.


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 Swing13 extends JFrame{
```

```
  //---------------------------------------//

  public static void main(String args[]) {
      new Swing13();
  }//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()
  //---------------------------------------//

  Swing13(){//constructor

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

    getContentPane().add(makeLabel(
         "EtchedBorder",new EtchedBorder()));
    getContentPane().add(makeLabel(
       "BevelBorder RAISED",new BevelBorder(
                        BevelBorder.RAISED)));
    getContentPane().add(makeLabel(
       "BevelBorder LOWERED",new BevelBorder(
                       BevelBorder.LOWERED)));
    getContentPane().add(makeLabel(
         "EmptyBorder",new EmptyBorder(
                                  5,5,5,5)));
    getContentPane().add(makeLabel(
      "Compound, Empty + BevelBorder RAISED",
      new CompoundBorder(new BevelBorder(
      BevelBorder.RAISED),new EmptyBorder(
                                  5,5,5,5))));
    getContentPane().add(makeLabel(
     "Compound, Empty + BevelBorder LOWERED",
     new CompoundBorder(new BevelBorder(
     BevelBorder.LOWERED),new EmptyBorder(
                                  5,5,5,5))));

    getContentPane().add(makeLabel(
      "Compound, Empty + SoftBevelBorder " +
      "RAISED",
      new CompoundBorder(new SoftBevelBorder(
      SoftBevelBorder.RAISED),new EmptyBorder(
                                  5,5,5,5))));
    getContentPane().add(makeLabel(
      "Compound, Empty + SoftBevelBorder " +
      "LOWERED",
```

```
      new CompoundBorder(new SoftBevelBorder(
      SoftBevelBorder.LOWERED),
                new EmptyBorder(
                                  5,5,5,5))));
    getContentPane().add(makeLabel(
      "Compound, Empty + LineBorder",
      new CompoundBorder(new LineBorder(
      Color.blue,5),new EmptyBorder(
                                  5,5,5,5))));
    getContentPane().add(makeLabel(
      "Compound, Empty + MatteBorder Image",
      new CompoundBorder(new MatteBorder(
      19,19,19,19,new ImageIcon(
      "blue-ball.gif")),new EmptyBorder(
                                  5,5,5,5))));


    getContentPane().add(makeLabel(
      "Compound, Empty + MatteBorder Color",
      new CompoundBorder(new MatteBorder(
               19,19,19,19,Color.blue),
                 new EmptyBorder(5,5,5,5))));



    getContentPane().add(makeLabel(
      "Compound, Empty + TitledBorder",
      new CompoundBorder(new TitledBorder(
        "Title"),new EmptyBorder(5,5,5,5))));


    setTitle("Copyright 2000, R.G.Baldwin");
    setSize(329,500);
    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 Swing13
```

Listing 3

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-