Swing from A to Z

The border Property

Part 4, LineBorder, MatteBorder, and TitledBorder

By Richard G. Baldwin

Java Programming, Lecture Notes # 1023

September 11, 2000

- Preface
- Introduction
- Sample Program
- Interesting Code Fragments
- Summary
- Where To From Here?
- Complete Program Listing

Preface

This lesson is Part 4 of a multi-part lesson 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

I recommend that you study The border Property, Part 1, The border Property, Part 2, and The border Property, Part 3 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 here.

Introduction

What's in this lesson?

This lesson illustrates the use of **LineBorder**, **MatteBorder**, and **TitledBorder**.

Two different variations of **MatteBorder** are illustrated. One uses a tiled icon to create the matte and the other uses a solid color.

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 below.

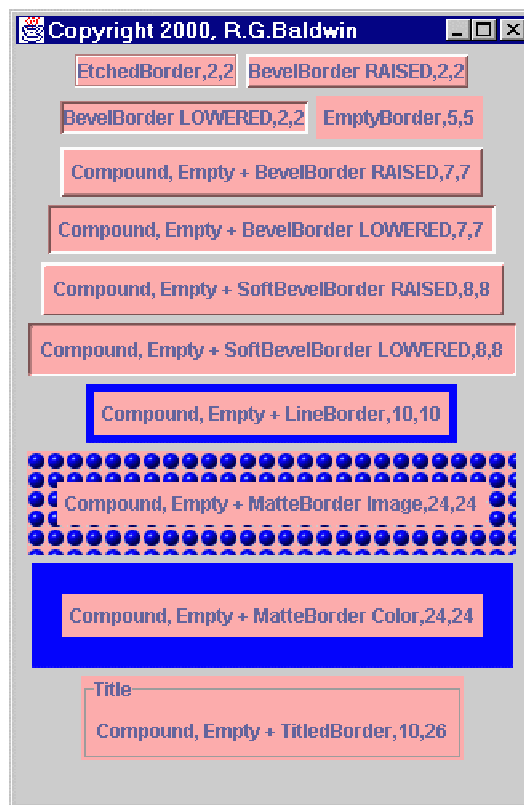

Figure 1, A Screen Shot of the Running Program

Twelve JLabel objects

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

In this lesson, I will be concentrating on the components appearing in the bottom four rows of the screen shot.

Interesting Code Fragments

I will discuss the program in fragments. A complete listing of the program is shown in Listing 5 near the end of the lesson. I will skip those parts of the program that were discussed in previous lessons

Compound Empty + LineBorder,10,10

Listing 1 shows the code fragment that caused the label in the seventh row of the screen shot to be added to the *contentPane* of the **JFrame** container. This is the label with the narrow blue border showing.

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

Listing 1
```

Familiar syntax

By now, you are familiar with (and possibly bored with) this syntax, so I won't discuss it in detail.

Briefly, this fragment uses a **CompoundBorder** object to combine a **LineBorder** object and an **EmptyBorder** object to form the visible border.

As in the previous lesson, the **EmptyBorder** is used to provide a blank margin between the text and the **LineBorder**.

What does Sun have to say?

Here is what Sun has to say about **LineBorder**.

> A class which implements a line border of arbitrary thickness and of a single color.

The constructors

Two simple overloaded constructors are available. Each let you specify the color of the line. One defaults to a line thickness of one pixel. The other lets you specify the thickness of the line in pixels.

In this case, I specified a blue line with a thickness of five pixels.

The inset values were each ten pixels, resulting from the sum of the blank area and the width of the line.

Compound Empty + MatteBorder Image,24,24

Listing 2 shows the code fragment that caused the label in the eighth row of the screen shot to be added to the *contentPane* of the **JFrame** container. This is the ugly label with the blue balls in its border.

```
    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))));

Listing 2
```

This rather ugly component uses a **CompoundBorder** object to combine a **MatteBorder** object and an **EmptyBorder** object.

MatteBorder class

Here is what Sun has to say about **MatteBorder**.

> A class which provides a matte-like border of either a solid color or a tiled icon.

Although the **MatteBorder** class is easy to use from a technical viewpoint, it can be a challenge from a cosmetic viewpoint when icons are used to produce the border. As you can see, mine didn't turn out to be very pleasing.

Dimensions are very important

In order to design a component with a pleasing appearance, the designer would need to be very careful about the size of the icon, the insets of the outside **MatteBorder**, and the insets of the inside **EmptyBorder**. Otherwise, the result isn't balanced, as is the case in my example.

Compound Empty + MatteBorder Color,24,24

Listing 3 shows the code fragment that caused the label in the ninth row of the screen shot to be added to the *contentPane* of the **JFrame** container. This is the label with the wide blue border.

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

Listing 3

This is a very straightforward fragment. It uses **EmptyBorder** to place a five-pixel blank band around the original **JLabel** component.

Then it uses **MatteBorder** to place a 19-pixel blue matte around the blank band, producing an overall inset value of 24 pixels.

Compound Empty + TitledBorder,10,26

Here is what Sun has to say about **TitledBorder**.

> A class which implements an arbitrary border with the addition of a String title in a specified position and justification.
>
> If the border, font, or color property values are not specified in the constructor or by invoking the appropriate set methods, the property values will be defined by the current look and feel, using the following property names in the Defaults Table:
>
> - "TitledBorder.border"
> - "TitledBorder.font"
> - "TitledBorder.titleColor"

This can be a very complex component. The **TitledBorder** class has six overloaded constructors that provide a wide variety of options in the construction of the object.

Listing 4 shows the code fragment that caused the label in the bottom row of the screen shot to be added to the *contentPane* of the **JFrame** container.

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

Listing 4

As you can see, I elected to use the simplest form of constructor, which required only that I provide a string for the title.

Summary

So far, in this miniseries on borders, I have introduced you to each of the standard **Border** classes, and have illustrated one or more variations on each of them.

I have also mentioned that if the standard borders won't fulfill your needs, you can define your own borders by defining a class that implements the **Border** interface.

Where To From Here?

I have two more topics to cover before I leave this miniseries on borders: nested compound borders and the **BorderFactory** class. I will cover those topics in the next two lessons.

Complete Program Listing

A complete listing of the program is shown in Listing 5.

```
/*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 5

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-