Swing from A to Z

The border Property

Part 1, EtchedBorder

By Richard G. Baldwin

Java Programming, Lecture Notes # 1020

August 21, 2000

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.

A recommended lesson

Since this lesson deals with borders, I particularly recommend the lesson entitled *"**Swing, Hidden Buttons with Icons, Icon Images, Borders, Tool Tips, Nested Buttons, and Other Fun Stuff**."* That lesson illustrates some very interesting uses of borders with buttons to cause buttons to rise up from the surface when you point at them with the mouse.

Introduction

Properties, events, and methods

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

Default appearance and behavior

Because most Swing components extend **JComponent**, the properties, events, and methods defined in those classes provide the default appearance and behavior of most of the Swing components.

Understanding common properties, events, and methods

The next few lessons concentrate on 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, I will embark on a study of the additional specialized appearance and behavior associated with individual components.

What's in this lesson?

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

Several parts are needed

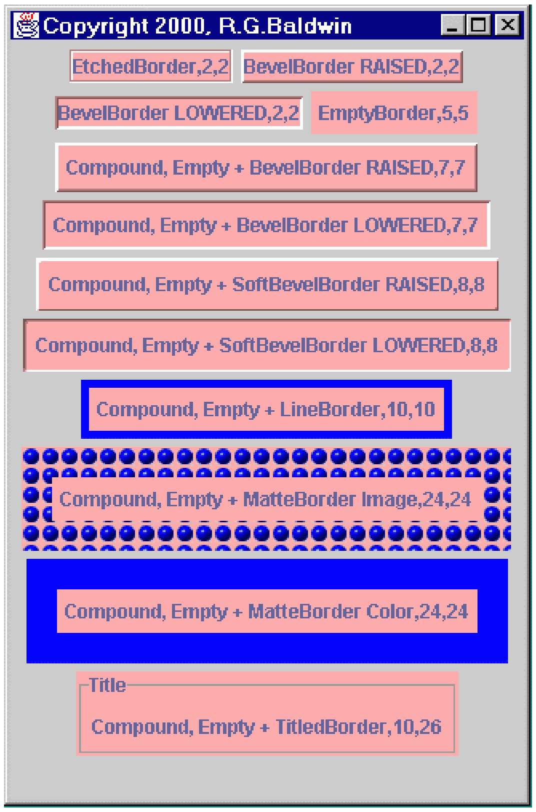
Because of the large amount of material involved, I have decided to break this discussion into several parts. This is Part 1. It sets the background for future discussions, and also deals specifically with the use of the **EtchedBorder** class.

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 run is shown below as Figure 1.


Copyright 2000, R.G.Baldwin
EtchedBorder,2,2
BevelBorder RAISED,2,2
BevelBorder LOWERED,2,2
EmptyBorder,5,5
Compound, Empty + BevelBorder RAISED,7,7
Compound, Empty + BevelBorder LOWERED,7,7
Compound, Empty + SoftBevelBorder RAISED,8,8
Compound, Empty + SoftBevelBorder LOWERED,8,8
Compound, Empty + LineBorder,10,10
Compound, Empty + MatteBorder Image,24,24
Compound, Empty + MatteBorder Color,24,24
Title
Compound, Empty + TitledBorder,10,26
Figure 1 Screen Shot of the GUI

Twelve JLabel objects

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

As you can see, a wide variety of possible border styles are available, and this is just a sampling of the possibilities.

Apply to all Swing components

These different border styles can be applied to all Swing components that extend **JComponent**, either directly or indirectly.

Can also define your own borders

If you want to, you can also define your own custom borders as well.

Interesting Code Fragments

I will discuss the program in fragments. A complete listing of the program is provided later as Listing 6.

The controlling class

Listing 1 shows the beginning of the controlling class and the **main()** method.

```
class Swing13 extends JFrame{

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

Listing 1

The controlling class extends **JFrame** so that an object of the controlling class is a GUI that can be placed directly on the desktop.

Instantiate an object of the controlling class

The **main()** method instantiates an object of the controlling class, causing the GUI to appear on the screen.

The makeLabel() method

Listing 2 shows the beginning of a convenience method named **makeLabel()**. This method is designed to instantiate and to return a reference to an opaque **JLabel** object with a specified border and a pink background.

```
  JLabel makeLabel(
          String content,Border borderType){

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

Listing 2

A convenience method

This method is provided to reduce the amount of code required to instantiate the twelve **JLabel** objects with borders. By using this method, it is possible to avoid the requirement to repeat the same code twelve times.

Text content of the label

The text content of the label is provided as the first incoming parameter to the method.

The required border style

The required border style is provided as the second incoming parameter. Note that this parameter is a reference to an object of type **Border**.

The new code

The only thing new in the code in Listing 2 is the use of the **setBorder()** method to set the *border* property of the label object. This setter method requires a parameter that is a reference to an object of a class that implements the **Border** interface (a reference to an object of type **Border**).

What does Sun have to say?

Here is what Sun has to say about the **setBorder()** method.

Sets the border of this component. The **Border** object is responsible for defining the insets for the component (overriding any insets set directly on the component) and for optionally rendering any border decorations within the bounds of those insets.

Borders should be used (rather than insets) for creating both decorative and non-decorative (e.g. margins and padding) regions for a swing component.

Compound borders can be used to nest multiple borders within a single component.

The insets

The term insets is a term that is used to describe the width of the border in pixels.

The **getInsets()** method returns the value of the *insets* property of the object on which the method is invoked. This value is returned as a reference to an object of the **Insets** class.

The Insets class

An object of the **Insets** class encapsulates four **public** fields of type **int**. The four fields contain the widths in pixels of the left, right, bottom, and top sections of the border.

Because they are public, these field values can be accessed simply by joining the name of the field to the name of a reference to the **Insets** object. (No accessor method is required.)

The remainder of the makeLabel() method

Listing 3 shows the remainder of the method named **makeLabel()**.

```
    label.setText(content + ","
        +label.getInsets().left + ","
        +label.getInsets().top);
    return label;
  }//end makeLabel()
```

Listing 3

The **getInsets()** method is used to get the inset values for the left border and top border. These two values are concatenated with the specified text content so that they will be displayed as the text on the label when the label is rendered.

Returns a JLabel object

The method returns a reference to a **JLabel** object, with an opaque pink background, having the specified borders, and having a *text* property whose value is the concatenation of the specified text value and the inset values for the left and top borders.

The constructor

Listing 4 shows the beginning of the constructor, which sets the layout manager to **FlowLayout**.

```
  Swing13(){//constructor
    getContentPane().setLayout(
                      new FlowLayout());
```

Listing 4

Will honor preferred size of the labels

As mentioned in an earlier lesson, this layout manager will attempt to honor both dimensions of the preferred size while displaying each label.

In this case, the value of the *preferredSize* property is automatically set, taking the borders and the text content of the label into account.

The rest of the constructor

Following this, the constructor contains twelve fairly complex statements. Each of the twelve statements causes a **JLabel** object to be displayed with

- A specified border style,
- The specified text content, and
- The left and top insets for the specified border style.

Open in another browser window

At this point, it may be helpful for you to open a copy of this lesson in another browser window so you can see the screen shot of the GUI while I discuss the different border styles.

EtchedBorder,2,2

Listing 5 shows the code fragment that causes the label in the upper-left corner of the screen shot to be added to the **JFrame** container.

```
    getContentPane().add(makeLabel(
        "EtchedBorder",new
EtchedBorder()));
```

Listing 5

Uses the makeLabel() method

This fragment invokes the **makeLabel()** method discussed above, passing the two required parameters.

The specified text content

The first parameter is the string "EtchedBorder" that is to become part of the text content of the label.

As you will recall, this string is concatenated with the inset values for the border and set into the *text* property for the label. This, in turn, causes the concatenated string to be displayed as the text on the face of the label. For this case, the resulting text on the face of the label is:

EtchedBorder,2,2

The specified border style

The second parameter to the **makeLabel()** method is a reference to a new object of the class **EtchedBorder**. This class (as well as the other classes that I will discuss in the following paragraphs) extends the class named **AbstractBorder**.

AbstractBorder

The **AbstractBorder** class implements the **Border** interface. Therefore, a reference to an object of the **EtchedBorder** class satisfies the requirement that the second parameter to the **makeLabel()** method be a reference to an object of type **Border**.

Creating your own borders

If you decide to create your own custom borders, probably the best way to do so is to extend the **AbstractBorder** class.

Now back to EtchedBorder

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

> A class which implements a simple etched border which can either be etched-in or etched-out.
>
> If no highlight/shadow colors are initialized when the border is created, then these colors will be dynamically derived from the background color of the component argument passed into the paintBorder() method.

The default case

The default case for the *noarg* constructor that I used is to produce a border that appears to be etched or chiseled out of the surface (a ditch). It has a very modest three-dimensional appearance created using highlights and shadows.

Other constructors available

As you can probably surmise from the Sun text provided above, there are other constructors that allow you to cause the border to appear as a small hill instead of a ditch. It is also possible to specify the colors used for highlights and shadows as well.

The size of the label

As you can see from the screen shot, the default size of the label object with the etched border is barely large enough to accommodate its text, particularly on the ends. All in all, it is pretty ugly.

We will see how to remedy this situation later by using a compound border.

The inset values

As you can also see from the text displayed in the label in the screen shot, the inset (width) of the left border and the inset of the top border are each two pixels.

Although I didn't display the inset for the right border and the bottom border, they appear to be the same for this border style.

Summary

In this lesson, I have introduced you to the general concept of applying borders to Swing components. I have illustrated the specific border style known as an **EtchedBorder**.

Where To From Here?

I will discuss two different versions of the border style known as **BevelBorder** in the next lesson.

Complete Program Listing

A complete listing of the program is provided in Listing 6. The purpose of this lesson was to introduce you to the use of borders in Swing. As you saw from the screen shot, this program produces a large number and variety of border styles. I have discussed only a small part of the program, along with only one border style in this lesson. I will continue discussing the program is future lessons.

```
/*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 6

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-