Swing from A to Z: Demystifying Glue and Struts, Part 3

Baldwin shows you how to use custom methods that provide the functionality of glue and struts. One of the methods produces an elastic spacer component with an upper limit on how far the component will stretch. Another method provides the functionality of glue and struts in a single object. Just for fun, Baldwin also shows you how to use a JButton as a container.

Published: January 2, 2001
By Richard G. Baldwin

Java Programming, Lecture Notes # 1036

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 understand Swing at a detailed level.

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.

Recommended supplementary reading

In the lesson entitled *Alignment Properties and BoxLayout, Part 1*, I recommended a list of Swing tutorials for you to study prior to embarking on a study of this set of lessons.

The lessons identified on that list will introduce you to the use of Swing while avoiding much of the detail included in this series.

Where are they located?

You will find those lessons published at Gamelan.com. However, as of the date of this writing, Gamelan doesn't maintain a consolidated index of my Java tutorial lessons, and sometimes my lessons are difficult to locate there. You will find a consolidated index at *Baldwin's Java Programming Tutorials*.

The index on my site provides links to the lessons at Gamelan.com.

Preview

In the previous lesson entitled *Swing from A to Z: Demystifying Glue and Struts, Part 2,* I developed three convenience methods that can be used as alternatives to the factory methods of the **Box** class to produce components that fulfill the functionality of glue and struts.

One of the methods makes it possible to produce an elastic spacer component (glue) with an upper limit on how far the component will stretch.

Another method returns a reference to an object that provides the functionality of glue and struts in a single object.

In this lesson, I will show you how to use the three methods mentioned above to complete the program that produced the screen shots shown in that lesson. *(Those screen shots are repeated in this lesson for convenience of viewing.)*

The program will also demonstrate that **BoxLayout** is not confined to being used with a **Box** container. Just for fun, I will use a **JButton** as a container with a **BoxLayout** manager.

Introduction

In an earlier lesson entitled *Swing from A to Z: Alignment Properties and BoxLayout, Part 1,* I introduced you to the **BoxLayout** manager. I will use **BoxLayout** in the sample program in this lesson.

Glue and struts

In another earlier lesson entitled *Swing from A to Z: Glue, Struts, and BoxLayout,* I introduced you to the use of glue and struts.

The previous lesson

In the previous lesson entitled *Swing from A to Z: Demystifying Glue and Struts, Part 2,* I developed three new methods for producing invisible components that can take the place of the glue and struts produced by the standard factory methods.

What do the new methods do?

These new methods provide alternative *glue* and *strut* components that have more functionality than the standard glue and struts.

How do they differ?

These alternative methods provide the following behaviors:

- You can change the size of the alternative *strut* component after the object is instantiated.
- You can specify a maximum stretch value for the alternative *glue* component.
- You can instantiate a single object that replaces the combination of a *glue* object and a *strut* object.

Discussed in detail earlier

I discussed the rationale behind and the behavior of those methods in detail in the previous lesson entitled *Swing from A to Z: Demystifying Glue and Struts, Part 2*.

In that lesson, I began the development of a program that uses the new methods, and showed you some screen shots produced by that program.

I promised that this lesson would complete the development of that program.

Sample Program

This sample program, named **Swing20**, is designed to take the mystery out of glue and struts as used with **BoxLayout.**

Figure 1 is a screen shot that shows what the screen looks like when this sample program starts running.

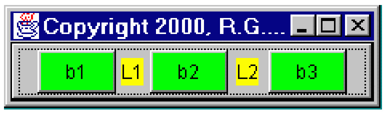

Figure 1. Screen shot when program starts running.

Must resize to see the effect

You must manually resize the **JFrame** object to make it larger and smaller in order to see the effect of the spacer components used in the program.

Figure 2 is a screen shot showing the effect of manually resizing the **JFrame** to make it narrower.

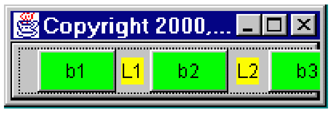

Figure 2. A screen shot after making the frame narrower.

Figure 3 is a screen shot showing the effect of making the frame wider.

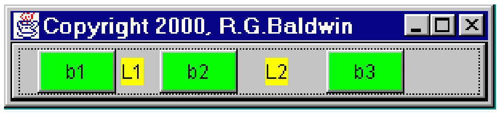

Figure 3. A screen shot after making the frame wider.

I discussed these screen shots at length in the previous lesson entitled *Swing from A to Z: Demystifying Glue and Struts, Part 2*.

Interesting Code Fragments

The name of this program is **Swing20**.

I will break this program down and discuss it in fragments. A listing of the entire program is provided in Listing 6.

The constructor

The beginning of the constructor is shown in Listing 1.

```
  Swing20(){//constructor

    JButton aBut = new JButton();
    aBut.setLayout(
       new
BoxLayout(aBut,BoxLayout.X_AXIS));

    //Add the JButton to the
contentPane
    getContentPane().add(aBut);

Listing 1
```

As I have mentioned several times in previous lessons, (almost) every Swing component is a container.

BoxLayout not confined to Box container

One of my objectives in this lesson is to demonstrate that **BoxLayout** is not confined for use only with the **Box** container.

Therefore, just for fun, I decided to use a large **JButton** as a container, and to apply a **BoxLayout** manager to that container. *(If you look very carefully at Figure 1, you will see that a large JButton object contains three small JButton objects and two JLabel objects. The small buttons appear to protrude from the large button.)*

The code to accomplish this is shown in Listing 1.

BoxLayout constructor is unusual

The constructor for a **BoxLayout** object is rather unusual, as layout managers go. Here is a description of the constructor for **BoxLayout**.

```
public BoxLayout(Container target,
             int axis)
```

Creates a layout manager that will lay out components either left to right or top to bottom, as specified in the axis parameter.

Parameters:

- target - the container that needs to be laid out
- axis - the axis to lay out components along. For left-to-right layout, specify BoxLayout.X_AXIS; for top-to-bottom layout, specify BoxLayout.Y_AXIS

Two parameters required

The constructors for many layout managers don't require any parameters. However, this constructor requires two parameters.

First parameter is controlled object

The first parameter must be a reference to the container object that will be under control of the layout manager. In this case, this parameter is a reference to the **JButton** object that serves as the container.

Second parameter specifies horizontal or vertical

The second parameter is a numeric value that specifies whether the layout will be along a horizontal or vertical axis. Symbolic constants are provided in the **BoxLayout** class for both of these possibilities.

Add to the content pane

The code in Listing 1 also adds the **JButton** container to the content pane, which is normal when using Swing.

Buttons and labels

The code in Listing 2 instantiates three **JButton** objects and two **JLabel** objects that will be placed in the large **JButton** container object.

```
    //Instantiate three JButton
objects,
    // make them green.
    JButton but1 = new JButton("b1");
    but1.setBackground(Color.green);

    JButton but2 = new JButton("b2");
    but2.setBackground(Color.green);

    JButton but3 = new JButton("b3");
    but3.setBackground(Color.green);

    //Instantiate two JLabel
objects.Color
    // them yellow.
    JLabel lab1 = new JLabel("L1");
    lab1.setBackground(Color.yellow);
    lab1.setOpaque(true);

    JLabel lab2 = new JLabel("L2");
    lab2.setBackground(Color.yellow);
    lab2.setOpaque(true);

Listing 2
```

(These buttons and labels will be separated by fixed and elastic spacer objects.)

The code in Listing 2 is straightforward, so I will let the comments speak for themselves.

Construct the GUI

The code in Listing 3 adds the buttons and the labels to the **JButton** container and inserts invisible spacer objects between them.

```
    aBut.add(but1);
    aBut.add(myFixedSpacer(3,0));
    aBut.add(lab1);
    aBut.add(myFixedSpacer(4,0));
    aBut.add(myElasticSpacer(6,0));
    aBut.add(but2);
    aBut.add(myElasticSpacer(12,0));
    aBut.add(myFixedSpacer(5,0));
    aBut.add(lab2);
    aBut.add(combinationSpacer(6,0,24,0));
    aBut.add(but3);

Listing 3
```

Insert a fixed-width spacer

The second statement in Listing 3 invokes the **myFixedSpacer()** method to insert a fixed spacer that is three pixels wide and zero pixels tall between the left-most button **b1** and the label to its right **L1**. *(See Figure 1 to identify the components labeled b1 and L1 that I am referring to.)*

The method named **myFixedSpacer()**, along with the other two methods discussed below, was described in detail in the previous lesson entitled *Swing from A to Z: Demystifying Glue and Struts, Part 2*.

Another fixed-width spacer

The second boldface statement in Listing 3 above invokes the same method to insert a fixed-width spacer four pixels wide between the label **L1** and the center button **b2**. *(Note that L1 and b2 refer to the captions on the components and not to the names of the variables that refer to those components.)*

An elastic spacer

The third boldface statement in Listing 3 invokes the **myElasticSpacer()** method to insert an elastic spacer between the fixed spacer described in the previous paragraph and the center button **b2**. *(Until stretched, this spacer has zero width and therefore, doesn't occupy any space on the screen.)*

In this case, the elastic spacer has a maximum stretch limit of six pixels in the horizontal and zero pixels in the vertical.

Another pair of spacers

Using the same methodology, the fourth and fifth boldface statements in Listing 3 insert an elastic spacer and a fixed spacer immediately to the right of the center button, **b2**.

In this case the horizontal stretch limit of the elastic spacer is twelve pixels and the horizontal size of the fixed spacer is five pixels.

A combination spacer object

Finally, the sixth boldface statement in Listing 3 invokes the **combinationSpacer()** method to insert a single spacer component immediately to the left of the right-most button, **b3**.

This single component functions as both strut and glue. It has a minimum width of six pixels on the horizontal and a maximum stretch of 24 pixels on the horizontal.

A short side trip

In the previous lesson entitled *Swing from A to Z: Demystifying Glue and Struts, Part 2*, I showed you a screen shot produced by a different program named **Swing21** that used the **combinationSpacer()** method exclusively for inserting spacers. The corresponding code fragment from that program is shown in Listing 4.

```
    aBut.add(but1);
    aBut.add(combinationSpacer(3,0,0,0));
    aBut.add(lab1);
    aBut.add(combinationSpacer(4,0,6,0));
    aBut.add(but2);
    aBut.add(combinationSpacer(5,0,12,0));
    aBut.add(lab2);
    aBut.add(combinationSpacer(6,0,24,0));
    aBut.add(but3);

Listing 4
```

A repeat of that screen shot follows. Note that it has been manually stretched to cause all of the spacers to reach their stretch limit.

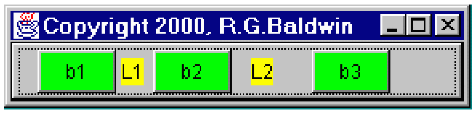

Figure 4. A screen shot using only combination spacers.

A fixed-width spacer

The first boldface statement in Listing 4 creates and inserts a spacer with a minimum width of three pixels. *(Because the specified maximum width is less than the minimum width, the method automatically sets the maximum width equal to the minimum width.)*

This produces a fixed-width spacer three pixels wide.

The remaining spacers

The last boldface statement in Listing 4 creates and inserts a spacer with a minimum width of six pixels and a maximum width of 24 pixels immediately to the left of **b3** (see Figure 4).

The boldface statements in between provide combination spacers on each side of **b2**.

Again, this code fragment came from a different program named **Swing21**, and produced the screen shot shown in Figure 4. *(See the previous lesson entitled Swing from A to Z: Demystifying Glue and Struts, Part 2, for a discussion of the purpose of this approach.)*

Now back to the main thread

The code in Listing 5 sets the look and feel to *Windows*.

```
    setTitle("Copyright 2000, R.G.Baldwin");

    //Set the look and feel.
    // First establish the string constants
    String metal =
      "com.sun.java.swing.plaf.metal." +
      "MetalLookAndFeel";
    String motif =
      "com.sun.java.swing.plaf.motif." +
      "MotifLookAndFeel";
    String windows =
      "com.sun.java.swing.plaf.windows." +
      "WindowsLookAndFeel";
    //Set the Look and Feel by enabling one
    // of these
    //String plafClassName = motif;
    //String plafClassName = metal;
    String plafClassName = windows;
    try{
      UIManager.setLookAndFeel(
                                plafClassName);
    }catch(Exception ex){
                         System.out.println(ex);}

    //Cause the L&F to become visible.
    SwingUtilities.
                  updateComponentTreeUI(this);
```

Listing 5

I have written numerous earlier tutorial lessons dealing with setting the look and feel, so I won't go into detail here. You will find those lessons at Gamelan.com, with links to the lessons in the Table of Contents on my web site.

You can experiment with the look and feel

I provided enough information in the three boldface statements in Listing 5 to allow you to experiment with different **lookAndFeel** property values.

How to change the look and feel

All that you need to do to see a different look and feel is to cause any one of the three boldface statements to be enabled, and to cause the other two boldface statements to be disabled by the comment indicators. Then recompile and run the program.

The remaining code in the program is too straightforward to merit discussion here. You can view that code in Listing 6.

Summary

So, there you have it, more than you probably ever wanted to know about the **Box** container, the **BoxLayout** manager, the **Box.Filler** class, glue, struts, and custom alternatives to glue and struts.

That wraps up my miniseries on this set of topics related to **BoxLayout**.

What's Next?

One of the major strengths of Java is the ability to use introspection for the analysis of components at runtime.

The next lesson in the *Swing from A to Z* thread will be the first lesson in a new miniseries on the use of introspection to analyze Swing components. This series will develop a very useful program that you can use to supplement the Java documentation package.

Complete Program Listing

A complete listing of the program is provided in Listing 6.

```
/*File Swing20.java
Rev 8/14/00
Copyright 2000, R.G.Baldwin

Illustrates use of invisible fixed-width and
elastic spacers to control the separation
between components.  In order to see the
effect of the elastic spacers, you must
manually resize the JFrame object to make
it larger and smaller.

Also demonstrates that BoxLayout can be used
with containers other than Box.

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 Swing20 extends JFrame{

  public static void main(String args[]) {
      new Swing20();
  }//end main()
  //---------------------------------------//

  //The following three methods create and
  // return invisible spacer objects.

  //Creates and returns a fixed spacer
  Box.Filler myFixedSpacer(int x, int y){
    return new Box.Filler(
      new Dimension(x,y),
      new Dimension(x,y),
      new Dimension(x,y));
  }//end myFixedSpacer()

  //Creates and returns an elastic spacer
  Box.Filler myElasticSpacer(int x, int y){
    return new Box.Filler(
      new Dimension(0,0),
      new Dimension(0,0),
      new Dimension(x,y));
  }//end myElasticSpacer()

  //Creates and returns an elastic spacer.
  // Another approach.
  JLabel combinationSpacer(
        int xmin,int ymin,int xmax,int ymax){
    //xmax should never be less than xmin
    //ymax should never be less than
ymin
    int tempX = (xmax < xmin)?xmin:xmax;
    int tempY = (ymax < ymin)?ymin:ymax;

    JLabel temp = new JLabel();
    temp.setMinimumSize(
                   new Dimension(xmin,ymin));
    temp.setPreferredSize(
                   new Dimension(xmin,ymin));
    temp.setMaximumSize(
                   new
Dimension(tempX,tempY));
    return temp;
  }//end combinationSpacer

  Swing20(){//constructor

    //Just for fun, instantiate a new
    // JButton object and use it as a
    // container.  Set its layout manager
    // to BoxLayout.
    JButton aBut = new JButton();
    aBut.setLayout(
       new BoxLayout(aBut,BoxLayout.X_AXIS));

    //Add the JButton to the contentPane
    getContentPane().add(aBut);

    //Instantiate three JButton objects,
    // make them green.
    JButton but1 = new JButton("b1");
    but1.setBackground(Color.green);

    JButton but2 = new JButton("b2");
    but2.setBackground(Color.green);

    JButton but3 = new JButton("b3");
    but3.setBackground(Color.green);

    //Instantiate two JLabel objects.Color
    // them yellow.
    JLabel lab1 = new JLabel("L1");
    lab1.setBackground(Color.yellow);
    lab1.setOpaque(true);

    JLabel lab2 = new JLabel("L2");
    lab2.setBackground(Color.yellow);
    lab2.setOpaque(true);

    //Add the buttons and the labels to the
    // Box.  Insert spacers between them.
    aBut.add(but1);
    aBut.add(myFixedSpacer(3,0));
    aBut.add(lab1);
    aBut.add(myFixedSpacer(4,0));
    aBut.add(myElasticSpacer(6,0));
    aBut.add(but2);
    aBut.add(myElasticSpacer(12,0));
    aBut.add(myFixedSpacer(5,0));
    aBut.add(lab2);
    aBut.add(combinationSpacer(6,0,24,0));
    aBut.add(but3);

    setTitle("Copyright 2000, R.G.Baldwin");

    //Set the look and feel.
    // First establish the string constants
    String metal =
      "com.sun.java.swing.plaf.metal." +
      "MetalLookAndFeel";
    String motif =
      "com.sun.java.swing.plaf.motif." +
      "MotifLookAndFeel";
    String windows =
      "com.sun.java.swing.plaf.windows." +
      "WindowsLookAndFeel";
    //Set the Look and Feel by enabling one
    // of these
    //String plafClassName = motif;
    //String plafClassName = metal;
    String plafClassName = windows;
    try{
      UIManager.setLookAndFeel(
                                plafClassName);
    }catch(Exception ex){
                      System.out.println(ex);}

    //Cause the L&F to become visible.
    SwingUtilities.
                 updateComponentTreeUI(this);

    //Pack the JFrame down around the
    // components
    pack();
    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 Swing20
```

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-