Swing from A to Z: Demystifying Glue and Struts, Part 2

Published December 18, 2000
By Richard G. Baldwin

Java Programming, Lecture Notes # 1035

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 this lesson, I will develop 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.

These methods will differ from the factory methods in several important respects.

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

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

Introduction

The BoxLayout manager

In the lesson entitled *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 the lesson entitled *Swing from A to Z, Glue, Struts, and BoxLayout*, I introduced you to the use of glue and struts. At that point in time, they were pretty mysterious. All we knew was that there were some factory methods that would return references to different types of invisible components that exhibited the behavior of glue and struts.

Invisible components were of type Component

The references were returned as type **Component**. We didn't really know the actual type of the components that were returned by the factory methods *(and also didn't need to care, except out of curiosity).*

The size properties

At that point in time, we didn't have the tools to unravel the mystery, because we hadn't yet studied the size properties that we studied later in the lesson entitled *Swing from A to Z, Minimum, Maximum, and Preferred Sizes*.

My earlier promise

In the previous lesson entitled *Swing from A to Z: Demystifying Glue and Struts, Part 1*, I introduced you to the output of a program that is designed to take the mystery out of glue and struts. I showed you some screen shots produced by that program. I promised that in this lesson, I would introduce you to the code from that program.

What are minimum, maximum, and preferred sizes?

All components that extend **JComponent** inherit the following three properties that support the use of layout managers:

- preferredSize
- minimumSize
- maximumSize

I discussed them at length in the lesson entitled *Swing from A to Z, Minimum, Maximum, and Preferred Sizes*.

Sample Program

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

Must resize to see the effect

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

BoxLayout can be used with many containers

This program also demonstrates that **BoxLayout** can be used with containers other than **Box**. This program uses a **JButton** as a container. The layout manager for the **JButton** is set to **BoxLayout.**

Some screen shots

The following three screen shots were discussed in the previous lesson entitled *Swing from A to Z: Demystifying Glue and Struts, Part 1*. I am presenting them again here for convenient viewing.

At startup

The following screen shot shows the output when you start the program. The large **JButton** contains three green **JButton** components, two yellow **JLabel** components, and several invisible spacer components.

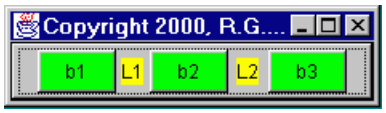

Figure 1. A screen shot at startup

When you make it narrower

Figure 2 shows the behavior of the **BoxLayout** manager with this set of components when the width of the container is decreased. The space between the components does not decrease, and the right-most component gets clipped at the edge of the **JButton**.

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

When you make it wider

Figure 3 shows what happens when you increase the width of the **JFrame**.

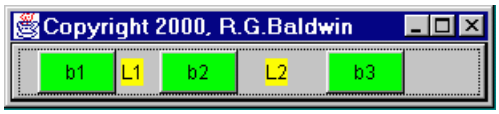

Figure 3. A screen shot after making the Frame wider.

Elastic spacers were used

Elastic spacers were inserted on both sides of the button labeled **b2**, and on the left side of the button labeled **b3**. These spacers have an upper limit on their elasticity, and once they reach that limit, they won't stretch any further. Once they quit stretching, the layout manager takes up the slack by inserting space at the right end of the large **JButton** container.

Interesting Code Fragments

As mentioned earlier, 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 5 near the end of the lesson.

The controlling class

Listing 1 shows the beginning of the controlling class and the **main()** method. You have seen code like this in several previous lessons. Therefore, I won't discuss it further here.

```
class Swing20 extends JFrame{

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

Listing 1
```

Significant new material

Listing 2 contains significant new material. This is the first of three listings that show the code for methods that return invisible spacer components of my own design.

```
  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()

Listing 2
```

A convenience method

The method that is shown in Listing 2 instantiates and returns a reference to an invisible fixed-size spacer component with specified width and height dimensions.

The width is specified by the incoming parameter named **x** and the height is specified by the incoming parameter named **y**.

An instance of the Box.Filler class

This method instantiates and returns a reference to an object of the **Box.Filler** inner class. *(If you are unfamiliar with inner classes, you might need to review my tutorials on that topic. Those tutorials are published at Gamelan.com. You will find a Table of Contents with links to those lessons at my personal web site.)*

What is the Box.Filler inner class?

Here is what Sun has to say about this class.

public static class **Box.Filler**
extends Component
implements Accessible

An implementation of a lightweight component that participates in layout but has no view.

A component that "has no view" is an invisible component. *(If you are unfamiliar with the concept of view, as embodied in the Model View Control (MVC) paradigm, I have previously written some tutorials on this topic as well.)*

Constructor for Box.Filler

This class has a single constructor that is described by Sun as follows:

public Box.Filler(Dimension min,
Dimension pref,
Dimension max)

Constructor to create shape with the given size ranges.

Parameters:
min - Minimum size
pref - Preferred size
max - Maximum size

Note the parameters

Our chief interest here lies in the three parameters. On the basis of your studies of the earlier lesson entitled *Swing from A to Z, Minimum, Maximum, and Preferred Sizes*, you should recognize that the **Dimension** objects passed as parameters to the constructor are used to set the value of the following properties of the **Box.Filler** object:

- minimumSize
- preferredSize
- maximumSize

Method to change the properties

Although I didn't use it in this program, there is also a method that can be used to modify these properties for an existing object of the **Box.Filler** class. That method is described below for the sake of completeness.

```
public void changeShape(
                    Dimension min,
                    Dimension pref,
                    Dimension max)

Change the size requests for this shape. An
invalidate() is propagated upward as a result so
that layout will eventually happen with using
the new sizes.

Parameters:
min - Value to return for getMinimumSize
pref - Value to return for getPreferredSize
max - Value to return for getMaximumSize
```

Now look at my code

Please refer back to Listing 2. The method named **myFixedSpacer()** receives a pair of **int** values that specify the width and height values for an invisible component.

Set three size properties to same value

The code in the method uses these two values to set the minimum, preferred, and maximum size properties of a **Box.Filler** object to the same size.

How does BoxLayout behave?

The behavior of the **BoxLayout** manager, when confronted with a component having the same values for all three properties, is to neither expand nor shrink that component during layout.

A fixed-size invisible component

Thus, the object that is returned by the method named **myFixedSpacer()** functions as an invisible component whose size is fixed when used with **BoxLayout**. This makes it useful for insertion between two visible components to create a fixed amount of blank space between them when used with **BoxLayout**.

Similar to Box.createHorizontalStrut()

Although I haven't investigated it down to the source code level, I believe that this is essentially the behavior of the factory method named **Box.createHorizontalStrut()** that was used in the earlier lesson entitled *Swing from A to Z, Glue, Struts, and BoxLayout* to get a *strut*.

In that case, there are separate factory methods provided for getting horizontal and vertical struts. However, I elected to combine both capabilities into a single method.

The mystery is solved

So now, the mystery is solved. The factory methods provided for obtaining struts and glue in the **Box** class probably return references to objects of the class named **Box.Filler**. If they don't, they could. *(I will leave it as an exercise for the reader to write and test the code to confirm or deny this hypothesis. A major hint on how to do that is provided later in this lesson.)*

What about type Component?

As mentioned earlier, the factory methods return references to the invisible components as type **Component**.

Since **Box.Filler** extends **Component**, and since we don't need to access any members of the objects returned by the factory methods such as **Box.createHorizontalStrut()**, it is satisfactory for the factory methods to return references to the struts and glue objects as type **Component**.

So what! you say

Although I believe that I have made it clear just what is going on with struts and glue, so far, I haven't done much to improve on the use of the available factory methods *(other than to make it possible to invoke changeShape() on the spacer object, which is not possible with a strut unless you downcast it to type Box.Filler).*

The next method does provide improvement

However, the code in Listing 3 does provide some improvement over the factory method named **Box.createGlue()**.

```
  //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()

Listing 3
```

No practical limit on stretch

When you invoke that factory method, you get back a glue object that that has no practical limit on how far it can stretch. *(I'm sure that there is a limit, but it is probably much greater than the width or height in pixels of any display monitor being manufactured today.)*

Upper limit on stretch

With the method named **myElasticSpacer()** in Listing 3, you can specify the maximum distance that the spacer will stretch in both directions.

Uses incoming parameters to set the limit

This is accomplished by using a pair of incoming parameters to establish the width and height values in the **maximumSize** property of the spacer object.

The **BoxLayout** manager won't cause a component to increase in size beyond this limit, so this provides an upper limit on the distance that the elastic spacer can be stretched.

Set minimum and preferred size to zero

Note that this method also sets the **minimumSize** and **preferredSize** properties to zero. Therefore, the normal behavior of the returned object is to occupy no space on the screen.

Clarification of an earlier lesson

I made the following statement in the earlier lesson entitled *Swing from A to Z, Glue, Struts, and BoxLayout*:

> "... the order in which the glue and strut components occur relative to each other is not important. In fact, they appear to occupy the same physical space regardless of the order in which you add them to the layout. (This appearance is an illusion. They don't really occupy the same physical space. I will explain this in more detail in a subsequent lesson.) "

The time has come to explain

The time has come to provide that explanation. The normal size of an elastic spacer (glue) is zero. Therefore, when it occupies the space on either side of a strut, there is no visible evidence that it even exists. It is not until it is stretched that its existence becomes apparent. At that point in time, the total space occupied by the strut and the glue is the sum of their respective sizes.

Another clarification

That also explains why doubling the maximum size of the elastic spacer to the left of **b3** in Figure 3 didn't completely double the size of the space to the immediate right of **b2**.

There were two spacers to the right of **b2**, but there was only a single combination spacer to the left of **b3**. *(A little later, we will see that the combination spacer doesn't add the minimum and maximum space.)*

The Combination Spacer

The method named **combinationSpacer()** in Listing 4 returns an invisible spacer of my own design that provides the functionality of a strut and some glue in a single object.

```
  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

Listing 4
```

Based on a transparent JLabel

A **JLabel** with no text is an invisible component as long as you don't make it opaque.

The method shown in Listing 4 returns a reference to a transparent **JLabel** object.

Size properties control its behavior

The preferred, minimum, and maximum property values for the object are set so as to cause it to act as a fixed size spacer on the **minimumSize** end, and to act as an elastic spacer up to the **maximumSize** end.

The **preferredSize** property is set equal to the **minimumSize** property so that it will display at the minimum size unless it is stretched by the layout manager.

Code is straightforward

The code in this method is straightforward, and shouldn't require an explanation. *(Note that because of the behavior of BoxLayout, the values for xmax and ymax should always be as large as the values of xmin and ymin.)*

Used for first three images

An object returned by this method was inserted immediately to the left of **b3** in the first three images shown above.

As mentioned earlier, even though the maximum stretch for this object was twice the maximum stretch of the elastic spacer immediately to the right of **b2**, the space produced wasn't twice as large. This is because the space to the right of **b2** also included a fixed-width spacer.

Use for all spacers

The screen shot shown in Figure 4 was produced using this combination method for all of the spacers *(rather than combining spacers of this type with pairs of fixed and elastic spacers).*

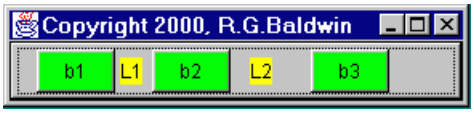

Figure 4. A screen shot using only combination spacers.

I believe that if you compare Figure 4 with Figure 3, you will see that each space in Figure 4 is more nearly double the space to its left *(as it should be)* than is the case in Figure 3.

This resulted from eliminating the sum of the fixed and elastic space values in the first three spaces from Figure 3. *(Just for my records, this screen shot was produced with a program named Swing21.)*

An important consideration

It is important for you to understand that the rendered size of a component in a GUI doesn't depend entirely on the size properties of the component. Rather, the rendered size also depends on how the layout manager treats the size properties.

Different layout managers behave differently

Different layout managers deal differently with the size properties. The **BoxLayout** manager honors the size properties whereas other layout manager may, or may not honor them.

Glue and struts work with BoxLayout

For that reason, glue and struts *(or alternatively the objects returned by my three methods)* behave according to their design with **BoxLayout**. They may not behave according to their design with other layout managers.

Summary

In this lesson, I have 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.

Different from the factory methods

These methods differ somewhat from the factory methods. In particular, 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 one of the methods returns a reference to an object that provides the functionality of glue and struts in a single object. Again, this object allows you to specify an upper limit on how far the object can stretch.

Screen shots

I also showed you some screen shots of a GUI produced using these three methods.

What's Next?

In the next lesson, I will show you how to use the three methods described above to complete the program that produced the screen shots shown in this lesson.

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

Complete Program Listing

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

```
/*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 5

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-