Custom Layouts

by Douglas Lyon
08/14/2003

In this article, we will show you how to roll your own layout manager. Java's *AWT* provides a series of layout manager classes that are used by both AWT and *Swing* for providing the service of *layout*. The layout service arranges components in a container by altering their size and location. In the `java.awt`, the `Component` class is subclassed by GUI elements with distinct behaviors, including `Button`, `Checkbox`, etc. In Swing, the `javax.swing.JComponent` is a subclass of the AWT's `java.awt.Component` and as such, inherits all the `Component` methods, as well as adding its own.

The layout service is provided by a series of classes in the `java.awt` and `javax.swing` packages. These classes use `setSize` and `setLocation` to arrange components in a container. A `Container` is a subclass of the `Component` class and is designed to hold component instances. For example, you can add `JButton`s to a `JPanel` and have them arranged by an instance of a layout class.

Note: It is unwise to program without a layout manager in place. Screens often have different resolutions.

The reason why layouts are used is that they enable flexibility in the size of the display. For example, if a user with a 640x480-pixel monitor wanted to use an application that was designed for a 1024x768 monitor, it would make sense to have the program adapt to the user (rather than have the user adapt to the program). In fact, even a change of font size can lead to a different layout, since the components that contain these fonts may well require a different amount of space to display them. Many users (including the author) would rather make use of a lower resolution setting on a large monitor in order to make the display easier to view. Some users have very small screens (i.e., those on palmtops or handhelds). Such displays do not have high resolutions. In fact, a 320x240-pixel resolution is high for some PDAs. As an added benefit, using a layout manager hides the complexity of layout from the programmer.

Programmers often find that they are using one of the three basic layouts in the `java.awt`. These are the `FlowLayout`, the `GridLayout`, and the `BorderLayout`. There are many such layouts, but these are among the most common and simple. These standard layouts are OK, but sometimes you want something just a bit different from a standard layout manager. For example, you might want to resize an image without altering the aspect ratio of the display, or you may find the layout manager too constraining. Beginners will often resort to turning the layout manager off by, for example, using:

```
setLayout(null);
```

Once the layout is turned off, size and placement of components is done without the benefit of the layout manager, using specific pixel locations and sizes. The argument often heard is "I want the component where I want it. The layout manager just gets in my way." This is a cry for a custom layout manager. The assumption that the screen size will be the same, so that a layout manager will not be needed, is without foundation.

The following sections review the `GridLayout` and show how to write your own custom layout manager, called the `PreferredSizeGridLayout`. The `PreferredSizeGridLayout` will never grow a component beyond its preferred size, but will shrink it below its minimum size if it does not fit in the available space.

Note: You must control your layout manager, or your layout manager will control you!

The `GridLayout`

In this section, we review the `GridLayout` and show its limitations and use.

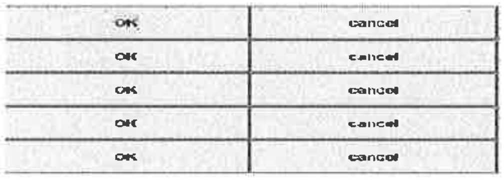
`GridLayout` is a class in the `java.awt` that arranges components in either a fixed number of rows or a fixed number of columns. If the programmer knows that there will be two columns of buttons, with an unknown number of rows, the GridLayout is created using:

```
new GridLayout(0,2);
```

The `0` means that the number of rows is unlimited.

The `GridLayout` is insensitive to the preferred size of its components. Normally an OK button takes up less room than a Cancel button, because the OK label in one button is smaller than the Cancel label in the other. The `GridLayout` style of layout makes all components take up the same amount of space. This might be inappropriate, depending on the look and feel you are trying to achieve.

Figure 1 shows the `GridLayout` with ten buttons. Each button takes up the same amount of space. The size of the label is ignored.



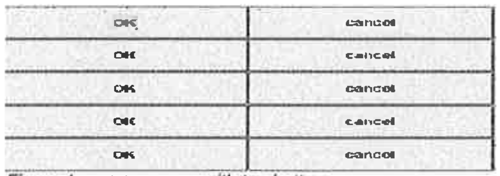


Figure 1. `GridLayout` *with ten buttons*

The code for Figure 1 follows:

```
public static void GridLayoutExample() {
      JFrame jf = new JFrame();
      Container c = jf.getContentPane();
      c.setLayout(new GridLayout(0, 2));
      c.add(new JButton("OK"));
      c.add(new JButton("cancel"));
      c.add(new JButton("OK"));
      c.add(new JButton("cancel"));
      c.add(new JButton("OK"));
      c.add(new JButton("cancel"));
      c.add(new JButton("OK"));
      c.add(new JButton("cancel"));
      c.add(new JButton("OK"));
      c.add(new JButton("cancel"));
      jf.setSize(200, 200);
      jf.setVisible(true);
}
```

The `PreferredSizeGridLayout`

The `GridLayout` is one of the standard layouts that programmers learn how to use early in their training. The `PreferredSizeGridLayout` presented below is just like a `GridLayout`; it just behaves differently. The `PreferredSizeGridLayout` pays attention to the preferred size of the component. In order to implement the custom layout manager, we create a hook via a callback method. To achieve polymorphism, we construct an interface so that creating a new manager of this type is easier in the future -- just implement the interface. Before we get into the details of its construction, let's take a look at why we might want a `PreferredSizeGridLayout` manager.

The best way to describe the difference between the `PreferredSizeGridLayout` manager and the `GridLayout` manager is to show an example:

```
public static void PreferredSizeGridLayoutExample() {
      JFrame jf = new JFrame();
      Container c = jf.getContentPane();
      c.setLayout(new PreferredSizeGridLayout(0,2));
      for (int i=0; i < 5; i++) {
          c.add(getOkButton());
          c.add(getCancelButton());
      }
      jf.setSize(200, 200);
      jf.setVisible(true);
}
public static void main(String args[]) {
      PreferredSizeGridLayoutExample();
}
public static JButton getOkButton() {
       JButton jb = new JButton("ok");
       jb.setMinimumSize(jb.getPreferredSize());
       jb.setMaximumSize(jb.getPreferredSize());
       return jb;
  }

public static JButton getCancelButton() {
       JButton jb = new JButton("cancel");
       jb.setMinimumSize(jb.getPreferredSize());
       jb.setMaximumSize(jb.getPreferredSize());
       return jb;
}
```

Figure 2 shows that the buttons do not expand to fill the available space, as in the `GridLayout`, but otherwise have the correct number of columns and rows.

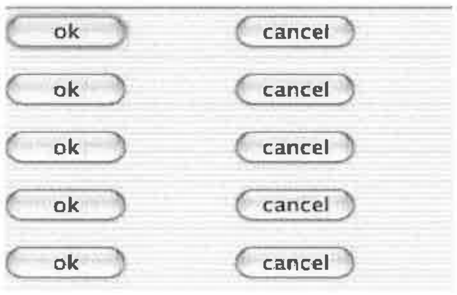


Figure 2. Buttons will not expand to fill available space.

Figure 3 shows that the buttons will shrink if the container gets too small to handle the preferred size of the buttons.

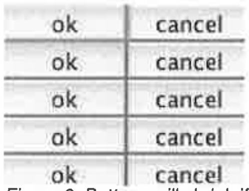


Figure 3. Buttons will shrink if container gets small.

To implement our new layout manager, we create a simple interface for controlling how a component is sized and placed:

```
package gui.layouts;

import java.awt.*;

public interface BoundableInterface {
      void setBounds(Component c, int x, int y, int w, int h);
}
```

Different applications will require different implementations of the `BoundableInterface`. For example, mailing labels must keep a fixed size, or they will not line up when printed. On the other hand, a large array of images can be made smaller and larger, but must keep the same width-to-height ratio (aspect ratio).

In the following implementation, we do not grow components beyond their preferred size to fill the cell of the layout:

```
package gui.layouts;

import java.awt.*;

public class PreferredBoundable
      implements BoundableInterface {
      public void setBounds(Component c,
                            int x, int y,
                            int w, int h) {
          Dimension wantedSize = new Dimension(w,h);
          Dimension d = c.getPreferredSize();
          // We select the minimum of the wantedSize
          // and the PreferredSize. Thus, the
          // component cannot grow beyond the
          // Preferred size, but it can shrink to
          // the wantedSize.
          d = min(d,wantedSize);
          c.setBounds(x,y,d.width,d.height);
      }

      public Dimension min(Dimension d1,
                           Dimension d2) {
          if (d1.width < d2.width) return d1;
          if (d1.height < d2.height) return d1;
          return d2;
      }
}
```

The `PreferredSizeGridLayout` manager follows. As an implementation of `LayoutManager`, it must implement a `layoutContainer` method, using size data provided by the components to perform layout. Most of the rest of its functionality is inherited from `GridLayout`. Notice that the implementation of `layoutContainer()` uses a `synchronized` call. Synchronizing on the object returned by the container's `getTreeLock` method ensures thread safety while performing the layout.

```
package gui.layouts;

import java.awt.*;

public class PreferredSizeGridLayout
      extends GridLayout {

      private BoundableInterface boundableInterface =
          new PreferredBoundable();

      public PreferredSizeGridLayout() {
          this(1, 0, 0, 0);
      }

      public PreferredSizeGridLayout(int rows,
                                     int cols) {
          this(rows, cols, 0, 0);
      }

      public PreferredSizeGridLayout(int rows,
                                int cols,
                                int hgap,
                                int vgap) {
    super(rows,cols,hgap,vgap);
}

/**
 * Lays out the specified container using this
 * layout.
 *
 * This method reshapes the components in the
 * specified target container in order to
 * satisfy the constraints of the
 *  PreferredSizeGridLayout object.
 *
 * The grid layout manager determines the size
 * of individual components by dividing the free
 * space in the container into equal-sized
 * portions according to the number of rows
 * and columns in the layout. The container's
 * free space equals the container's
 * size minus any insets and any specified
 * horizontal or vertical gap. All components
 * in a grid layout are given the Minimum of
 * the same size or the preferred size.
 *
 * @param target the container in which to do
 *       the layout.
 * @see java.awt.Container
 * @see java.awt.Container#doLayout
 */

public void layoutContainer(Container parent) {
    synchronized (parent.getTreeLock()) {
        Insets insets = parent.getInsets();
        int ncomponents =
            parent.getComponentCount();
        int nrows = getRows();
        int ncols = getColumns();

        if (ncomponents == 0) {
            return;
        }
        if (nrows > 0) {
            ncols = (ncomponents + nrows - 1) /
                    nrows;
        } else {
            nrows = (ncomponents + ncols - 1) /
                    ncols;
        }
        int w = parent.getWidth() -
               (insets.left + insets.right);
        int h = parent.getHeight() -
               (insets.top + insets.bottom);
        w = (w - (ncols - 1) * getHgap()) /
             ncols;
        h = (h - (nrows - 1) * getVgap()) /
            nrows;

        for (int c = 0, x = insets.left;
             c < ncols;
             c++, x += w + getHgap()) {
            for (int r = 0, y = insets.top;
                 r < nrows;
                 r++, y += h + getVgap()) {
                int i = r * ncols + c;
                if (i < ncomponents) {
                    boundableInterface.setBounds(
                        parent.getComponent(i),x, y,
                        w, h);
                }
            }
        }
    }
}

public BoundableInterface getBoundableInterface() {
    return boundableInterface;
}

public void setBoundableInterface(BoundableInterface
               boundableInterface) {
    this.boundableInterface = boundableInterface;
}

}
```

The programmer who uses the `PreferredSizeGridLayout` manager may override the default implementation of the `BoundableInterface` by invoking `setBoundableInterface` before the layout manager is used by the container. For example, to retain the rectangular shape and aspect ratio of components, as shown in Figure 4:

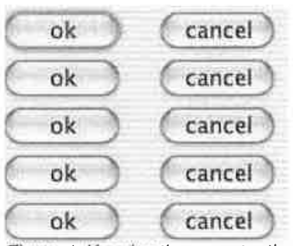


Figure 4. Keeping the aspect ratio in the grid

we tell the layout to use an `AspectBoundable`:

```
public static void PreferredSizeGridLayoutExample() {
      JFrame jf = new JFrame();
      Container c = jf.getContentPane();
      PreferredSizeGridLayout psgl =
          new PreferredSizeGridLayout(0, 2);
      psgl.setBoundableInterface(new AspectBoundable());
      c.setLayout(psgl);
      for (int i = 0; i < 5; i++) {
       c.add(getOkButton());
       c.add(getCancelButton());
      }
      jf.setSize(200, 200);
      jf.setVisible(true);
}
```

where `AspectBoundable` is given by:

```
package gui.layouts;

import java.awt.*;

public class AspectBoundable
      implements BoundableInterface {
      public void setBounds(Component c,
                            int x, int y,
                            int w, int h) {
          Dimension wantedSize =
              new Dimension(w, h);
          Dimension d = c.getPreferredSize();
          d = scale(d, wantedSize);
          c.setBounds(x, y, d.width, d.height);
      }

      /**
       * scale returns a new dimension that has
       * the same aspect ratio as the first
       * dimension but has no part larger than the
       * second dimension
       */
      public Dimension scale(Dimension imageDimension,
                             Dimension availableSize) {
          double ar =
             imageDimension.width /
             (imageDimension.height*1.0);
          double availableAr = availableSize.width /
             (availableSize.height*1.0);
          int newHeight =
              (int)(availableSize.width / ar);
          int newWidth =
              (int)(availableSize.height * ar);
          if (availableAr < ar )
              return new Dimension (availableSize.width,
                                    newHeight);
          return new Dimension(newWidth,
                               availableSize.height);
      }

      public Dimension scaleWidth(Dimension d1,
                                  Dimension d2) {
          double scaleFactor =
              d2.width / (d1.width * 1.0);
          return scale(d1, scaleFactor);
      }

      private Dimension scale(Dimension d1,
                              double scaleFactor) {
          return new Dimension(
                    (int) (d1.width * scaleFactor),
                    (int) (d1.height * scaleFactor));
        }

        public Dimension scaleHeight(Dimension d1,
                                     Dimension d2) {
            double scaleFactor = d2.height /
                                (d1.height * 1.0);
            return scale(d1, scaleFactor);
        }

}
```

Thus, we are able to obtain a modicum of control over our layout by implementing the `BoundableInterface`.

Summary

In this article, we showed how to take control of your own layout manager. These layout managers have been tested on multiple platforms and work on both Swing- and AWT-based GUIs.

The advantage of rolling your own layout manager is that you can get a lot more control over the appearance of your layout than if you use an existing layout manager. To learn more about layout managers and how to get the full software listing of the code in this article, see the DocJava web site. Also, be on the lookout for Lyon's new book, *Java for Programmers*, published by Prentice Hall.

Douglas Lyon is President of DocJava, Inc., a Java consulting firm in Connecticut specializing in Java training, and is the author of Prentice-Hall's upcoming book "Java For Programmers".

View all java.net Articles.

When might you use a custom layout manager?

Post Comment

XML java.net RSS